TECHNICAL SUMMARY

digital

VAX11 780

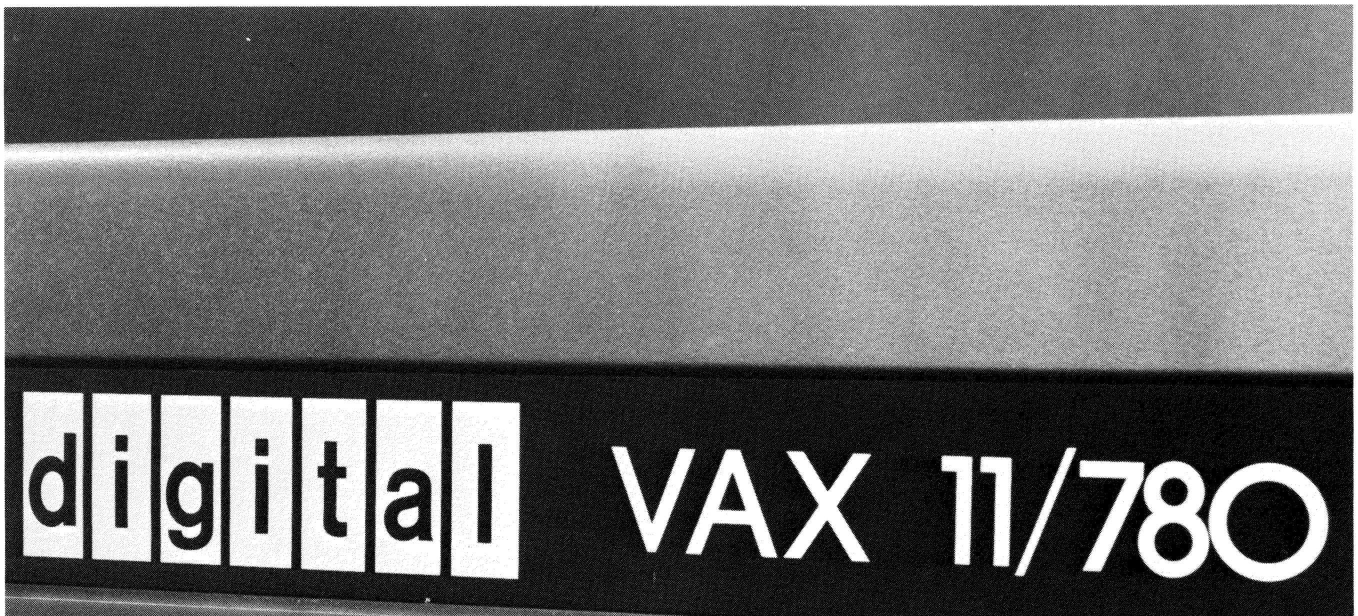
Contents

9 NETWORKS

10 SUPPORT SERVICES

11 GLOSSARY

1
Introduction



This technical summary introduces the characteristic features and capabilities of VAX-11/780 to system programmers and computer system specialists. Application programmers, managers, and operators can also use the technical summary to become familiar with the components, services, and operation of the system.

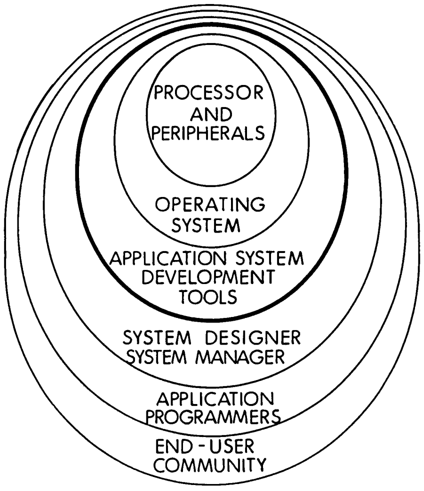
This document is a detailed technical introduction to all aspects of the VAX-11/780 system — from the VAX-11/780 processor and peripherals to the VAX/VMS operating system and DIGITAL's support services. The technical summary is primarily intended for the systems programmer or computer system specialist who is already familiar with computer hardware and software, but it contains useful information for application programmers, system managers, and computer operators.

You are encouraged to read the technical summary selectively. Many of the system's concepts and features are repeated throughout the text in different contexts. You might first skim through the summary to find those topics that interest you most, perhaps by reading just the abstracts that appear at the beginning of each section. You can then start with the sections of interest, knowing which section to refer to when you come across references to concepts discussed elsewhere.

If you are familiar with computer industry terminology and simply want an overview of the VAX-11/780 features, you should read the System section. The System section briefly identifies VAX-11/780 characteristics and introduces the features of the system that are described in detail throughout the remainder of the document.

If you are uncertain about what we mean by a particular term or phrase, you can probably find its definition in the glossary at the end of the technical summary. The glossary does not generally contain any standard computer-related terms, but it does contain most of the terms found throughout the VAX-11/780 documentation that have special meanings in the context of the VAX-11/780 system. For terminology not listed in the glossary, you should be able to use the definitions found in any generally-available dictionary of computer terms. The glossary is followed by a list of the mnemonics you may come across in the text. The mnemonics list is particularly useful while you are becoming familiar with the system.

Some people may find it more helpful to begin with the Users section. This section contains a description of the command language, from which you can gain an idea of the system's general capabilities. The Users section also introduces many of the aspects of the system that support applications programming, system management, and operator control. The section contains descriptions of potential applications that could be developed using the VAX-11/780. Almost all of the system features introduced in the Users section are further described in other sections of the summary.


PROCESSOR AND PERIPHERALS
OPERATING SYSTEM
APPLICATION SYSTEM DEVELOPMENT TOOLS
SYSTEM DESIGNER SYSTEM MANAGER
APPLICATION PROGRAMMERS
END - USER COMMUNITY

Users section are further described in other sections of the summary.

If you are a systems programmer familiar with an assembly language or software executive, you will find that the Processor and Operating System sections provide an in-depth discussion of the system's characteristics and capabilities. The concepts developed in both sections are closely related; for example, the respective discussions on memory management, I/O processing, and the compatibility mode environment in each section can be read together to gain a full appreciation for the system's design.

Perhaps one of the most important features of VAX-11/780 is that its programmers do not have to know assembly language to use the system effectively. Both the hardware and software contain many features that promote efficient and productive high-level language programming. If you are a high-level language programmer, you should glance through the Users, Languages, Data Management Services, and Network Services sections as appropriate to your interest in the system. The beginning of the Operating System section is also helpful if you want an introduction to some of the VAX-11/780 software terminology and concepts.

Finally, if you are considering obtaining a system, or have a system now, you should read the Support Services section to become familiar with the kinds of services available to our computer system users.

BLANK

2 The System

VAX-11/780 provides the performance, reliability, and programming features often found only in much larger systems. The processor has a 32-bit architecture based on the PDP-11 family of 16-bit minicomputers. While using addressing modes and stack structures similar to those of the PDP-11, the VAX-11/780 provides 32-bit addressing for a large program address space, and 32-bit arithmetic and data paths for processing speed and accuracy.

The processor's variable length instruction set and variety of data types, including decimal and character string, promote high bit efficiency. The processor hardware and instruction set specifically implement many high-level language constructs and operating system functions.

VAX-11/780 is a multiuser system for both program development and application system execution. It is a priority-scheduled, event-driven system: the assigned priority and activities of the processes in the system determine the level of service they need. Real-time processes receive service according to their priority and ability to execute, while the system manages CPU time and memory residency allocation for normal executing processes.

VAX-11/780 is a highly reliable system. Built-in protection mechanisms in both the hardware and software ensure data integrity and system availability. On-line diagnostics and error detecting and logging verify system integrity. Many hardware and software features provide rapid diagnosis and automatic recovery should the power, hardware, or software fail.

The system is both flexible and extendable. The virtual memory operating system enables the programmer to write large programs that can execute in both small and large memory configurations without requiring the programmer to define overlays or later modify the program to take advantage of additional memory. The command language enables users to modify or extend their command repertoire easily, and allows applications to present their own command interface to users.

INTRODUCTION

VAX-11/780 is a high performance multiprogramming computer system. The system combines a 32-bit architecture, efficient memory management, and a virtual memory operating system to provide essentially unlimited program address space.

The processor's instruction set includes integral floating point, packed decimal arithmetic, and character string instructions. Many of the instructions are direct counterparts for high-level language statements. The software system supports programming languages that take advantage of these instructions to produce extremely efficient code.

The VAX/VMS virtual memory operating system provides a multiuser, multilanguage programming environment on the VAX-11/780 hardware. The integral floating point instructions, efficient scheduler, and optional FORTRAN IV-PLUS language are ideal for real-time and scientific computational environments. The virtual memory capabilities, high-level languages, data management services, and large capacity peripherals make the system well suited to commercial applications. The system management facilities, command language, and program development tools provide the resources for general purpose applications development and execution. Spooling and extensive job control capabilities support batch processing.

The VAX-11/780 architecture is an extension of the PDP-11 family architecture. The processor executes variable length instructions in native mode, and non-privileged PDP-11 instructions in compatibility mode. Native mode includes the PDP-11 data types, and uses addressing modes and instructions similar to those of the PDP-11. The software supports compatible languages and file and record formats.

COMPONENTS

The major components of the VAX-11/780 are the:

- Processor — includes the basic CPU, a synchronous system bus, intelligent microcomputer console, interval and time of day clocks, and 8K bytes of cache. Up to 8 million bytes of MOS memory, two kinds of peripheral bus, and a floating point accelerator can be included with the processor.
- Peripherals — includes a range of small- and large-capacity disk drives, magnetic tape systems, hard copy and video terminals, line printers, and card readers.
- Operating System — includes a virtual memory manager, swapper, system services, device drivers, file system, record management services, command language, and operator's and system manager's tools.
- Languages — includes the VAX-11 MACRO assembly language and optionally VAX-11 FORTRAN IV-PLUS, VAX-11 COBOL-74, PDP-11 BASIC-PLUS-2/VAX, PDP-11 RPG II/VAX , and BLISS-32. Development tools for both native and compatibility mode programs include editors, linkers, librarians, and debuggers.
- Network Services — includes the DECnet/VAX network software and the DMC11 interprocessor communications link.

Processor

The VAX-11/780 processor provides 32-bit addressing, sixteen 32-bit general registers, and 32 interrupt priority levels. The instruction set operates on integer, floating point, character and packed decimal strings, and bit fields. The instruction set supports nine fundamental addressing modes.

The processor includes an 8K byte write-through memory cache that results in an effective 290-nanosecond memory access time. The processor's memory management includes four hierarchical processor access modes that are used by the operating system to provide read/write page protection between user software and system software.

Error Correcting Code (ECC) MOS memory is connected to the main control and data transfer path (called the SBI) via a memory controller. Physical memory is built using 16K MOS RAM chips and is organized in 72-bit words (64 bits for data and 8 for ECC). Each memory controller includes a request buffer that substantially increases overall system throughput and eliminates the need for interleaving in most applications.

The processor uses two standard clocks — a programmable real-time clock used by the operating system and by diagnostics, and a time-of-year clock used for system operations. The time-of-year clock includes battery back-up for automatic system restart operations.

The processor's console consists of an intelligent microcomputer (LSI-11) with 16K bytes of read/write memory and 8K bytes of ROM, a floppy disk, and a terminal for local operations and an optional port for remote diagnosis. The console operator uses keyboard commands for diagnosis, bootstrapping, and incorporating software maintenance modifications.

Medium capacity disks, unit record devices, terminals, the interprocessor communications links, and user-specific devices are UNIBUS peripherals. The UNIBUS adaptor provides the hardware pathways for data and control information to move between the UNIBUS and the SBI. The maximum aggregate throughput rate is 1.5 million bytes per second.

High-performance MASSBUS mass storage peripherals are connected to the SBI via a buffered MASSBUS adaptor. The MASSBUS adaptors provide the hardware pathways for data and control information to move between a MASSBUS peripheral controller and the SBI, and allow high-speed data transfers at a maximum aggregate throughput rate of 2 million bytes per second for each adaptor. The MASSBUS adaptor does parity checking for both data and control information.

Virtual Memory Operating System

VAX/VMS is a multiuser, multifunction virtual memory operating system that supports multiple languages, an easy to use interactive interface, and program development tools. The VAX/VMS operating system is designed for many applications, including scientific/real-time, computational, data processing, transaction processing, and batch.

The operating system performs process-oriented paging, which allows execution of programs that may be larger than the physical memory allocated to them. Paging is handled automatically by the system, freeing the user from any need to structure the program. In the VAX/VMS operating system, a process pages only against itself — thus individual processes cannot significantly degrade the performance of other processes.

The memory management facilities provided by the operating system can be controlled by the user. Any program can prevent pages from being paged out of its working set. With sufficient privilege, it can prevent pages from being swapped out, or prevent the entire working set from being swapped out, to optimize program performance for real-time or interactive environments. Sharing and protection are provided for individual 512-byte pages. Four hierarchical access modes (kernel, executive, supervisor, and user) provide page protection.

VAX/VMS schedules CPU time and memory residency on a preemptive priority basis. Thus, real-time processes do not have to compete with lower priority processes for scheduling services. Scheduling rotates among processes of the same priority. The scheduler adjusts the priorities of processes assigned one of the low 16 priorities to overlap I/O and computation. Real-time processes can be placed in one of the top 16 scheduling priorities, in which case the scheduler does not alter their priority. Their priorities can be altered by the system manager or an appropriately privileged user.

Interprocess communication is provided through shared files and shared address space, event flags, and mailboxes, which are record-oriented virtual devices.

VAX/VMS provides system management facilities. A system manager and a system operator are given the tools necessary to control the system configuration and the operations of system users for maximum efficiency.

The VAX/VMS command language is easy to learn and use, and is suitable for both the interactive and batch environments. Extensive batch facilities under VAX/VMS include job control, multistream spooled input and output, operator control, conditional command branching and accounting.

Command procedures are supported by the command languages as an easy method of executing strings of frequently used sequences of commands, or creating new commands from the existing command set. Command procedures accept parameters and can include extensive control flow.

VAX/VMS provides a program development capability that includes editors, language processors, and a symbolic debugger. The VAX-11 FORTRAN IV-PLUS, VAX-11 MACRO, BLISS-32, and the VAX-11 COBOL-74 language processors produce native code. The BASIC-PLUS-2/VAX and PDP-11 RPG II/VAX language processors produce compatibility mode code.

The VAX/VMS operating system provides a file and record management facility that allows the user to create, access, and maintain data files and records within the files with full protection. The record management services handle sequential, relative, and indexed file organizations, sequential and random record access, and fixed and variable-length records. Indexed files with sequential and random record access are available to compatibility mode programs, such as those written in PDP-11 RPG II/VAX or PDP-11 BASIC-PLUS-2/VAX.

The VAX/VMS operating system supports the Files-11 On-Disk Structure Level 2 (ODS-2), which provides the facilities for file creation, extension, and deletion with owner-specified protections and multilevel directories. On-Disk Structure Level 2 is upward compatible with Level 1, the file system currently available under the IAS and RSX-11 PDP-11 operating systems. Both native and compatibility mode programs can access both Level 1 and Level 2 volume structures.

DECnet/VAX networking capabilities are available as an

option for point-to-point interprocess communication, file access, and file transfer, and include down-line command file and RSX-11S system image loading.

PERFORMANCE

Many features of the VAX-11/780 ensure that the system provides real-time, computation, and transaction applications with the processing speed and throughput they need.

Processing

The VAX-11/780 hardware uses Schottky TTL logic circuits — a proven technology that combines fast switching speed with moderate power consumption. Emitter-coupled logic (ECL) circuits and large-scale integrated circuits have been used where appropriate to optimize system performance and reliability. The processor provides both 32-bit and 16-bit arithmetic, instruction pre-fetch, and an address translation buffer.

An optional high-performance floating point accelerator executes the standard floating point instructions and the multiply and polynomial evaluation instructions. The integer multiply instruction is useful in matrix manipulation subscript calculation, and the VAX/VMS operating system's mathematics library sine, cosine, and other functions make use of the polynomial evaluation instruction to significantly reduce the execution time of its subroutines.

The VAX-11/780 processor architecture is specifically designed to support high-level language programming. The high-level language compilers can produce efficient user programs because the instruction set is extremely bit-efficient. Among the features of the processor that serve to reduce program size and increase execution speed are the:

- variable length instruction format
- floating point, packed decimal, and character string data types
- indexed, short displacement, and program counter relative addressing modes
- small constant short literals

Furthermore, many instructions correspond directly to high-level language constructs:

- the Add Compare and Branch instruction for DO and FOR loop control
- the Case instruction for computed GO TO statements
- the 3-operand arithmetic instructions for statements such as "A = B+C"
- the Index instruction for computing index values, including subscript range checking
- the Edit instruction for output formatting

Much of the processor architecture also ensures that the operating system incurs minimal overhead for real-time multiprogramming. For example, the operating system uses:

- the context-switching instructions and queue instructions to schedule processes
- the asynchronous system trap (AST) delivery mechanism to speed returns from system service calls

Careful design, coding, and performance measurement ensure that the flow within the system is as rapid as possible.

Data Throughput

VAX-11/780 includes many features that support high data throughput, including silo data buffers for MASSBUS peripheral controllers, buffered direct memory access for the UNIBUS peripherals, and 64-bit data transfers and prefetching.

Memory bandwidth matches that of the processor's internal bus — 13.33 million bytes per second, including time for refresh cycles. This is primarily because of the memory controller request buffers, which substantially increase memory throughput and overall system throughput, and decrease the need for interleaving for most configurations. Memory interleaving, which is enabled and disabled under program control, can be used effectively when more than two MASSBUS peripheral controllers are connected and the MASSBUS and UNIBUS devices are transferring at very high rates — greater than one million bytes per second.

The operating system supports the hardware throughput in its I/O request processing software. The software uses the processor's multiple hardware priority levels to increase I/O response time, and keeps each disk controller as busy as possible by overlapping seek requests with I/O transfers.

RELIABILITY

Built-in reliablity features for both hardware and software provide data integrity, increased up-time, and fast system recovery from power, hardware, or software failures. Some of the many VAX-11/780 reliability features are discussed in the following paragraphs.

Data Integrity

VAX-11/780 provides memory access protection both between and within processes. Each process has its own independent virtual address space which can be mapped to private pages or shared pages. A process can not access any other process' private pages. The VAX/VMS operating system uses the four processor access modes to read and/or write protect individual pages within a process. Protection of shared pages of memory, files, and interprocess communication facilities such as mailboxes and event flags is based on file ownership and application group identification.

The VAX/VMS file system detects bad blocks dynamically and prevents re-use once the files to which they are allocated are deleted. The file system records critical information such as file header information to permit its recovery in the event of accidental destruction.

Integral fault detection hardware includes:

- Memory error correcting code that detects all double-bit errors and corrects all single-bit errors.
- Disk error correcting code that detects all errors up to 11 bits and corrects errors in a single burst of 11 bits.

- Extensive parity checking performed on the processor and peripherals buses, memory cache, address translation buffer, microcode, and writable diagnostic control store.
- Peripheral write-verify checking that checks all input and output disk and tape operations and ensures data reliability.
- Track offset retry hardware that enables the operating system to recover from disk transfer errors.

System Availability

The VAX/VMS operating system allows the VAX-11/780 system to continue running even though some of its hardware components have failed. The system automatically determines the presence of peripherals on the processor at bootstrap time. If the usual system device is unavailable, the system can be bootstrapped from any disk. If memory units are defective, memory is configured so that defective modules are not referenced. Software spooling allows output to be generated even if the normal output devices are not available.

The system operator can perform software maintenance activities without bringing the system down for stand-alone use. The operator can perform disk backup for both full volume backup/restore and single file backup/restore concurrent with normal activities.

The VAX/VMS operating system supports on-line peripherals diagnostics. VAX/VMS performs on-line error logging of CPU errors, memory errors, peripheral errors, and software failures. The operator or field service engineer can examine and analyze the error log file while the system is in operation.

System Recovery

Automatic system restart facilities bring up the system without operator intervention after a system failure caused by a power interruption, a machine check hardware malfunction, or a fatal software error. The VAX/VMS operating system automatically performs machine checks and internal software consistency checks during system operation. If the checks fail, VAX/VMS performs a system dump and reboots the system if the operator has set the system for auto-restart.

Memory battery backup can be used to preserve the contents of memory during a power outage. If memory battery backup is used, the time-of-year clock allows the recovery software to calculate elapsed time of the outage. A special memory configuration register indicates to the recovery software whether data in memory was lost. Following a power failure, the recovery software restarts all possible I/O in progress before the failure occurred. Programs can use the VAX/VMS power fail asynchronous system trap facility to initiate user-specific power fail recovery processing.

VAX-11/780 remote diagnosis allows DIGITAL field service engineers to run diagnostics, examine memory locations, and diagnose problems from a remote terminal. The engineer who goes to the site is prepared in advance to correct any problems that may have occurred.

FLEXIBILITY

VAX-11/780 is a system that is easy to use because it is both flexible and easy to extend. Some of the ways in which VAX-11/780 provides the user with flexible operating and programming environments are introduced below.

Flexibility in the Operating Environment

Virtual memory gives the user the ability to write and execute arbitrarily large programs without concern for addressing limitations. The paging and swapping algorithms allow more programs to execute than the available physical memory would allow if all programs had to be totally resident.

Both paging and swapping are transparent to the user, and therefore allow the system to be extended without reprogramming. The system's physical memory configuration can change without requiring program redesign or relinking. Programmers never have to structure their programs, although they can, at their option, to achieve maximum efficiency and performance for a given program. They can control working set size, lock pages in the working set or memory, and lock an entire working set in memory. In addition, the system manager can control the amount of time a process is guaranteed memory residency once it is swapped in.

The VAX/VMS scheduler recognizes 32 scheduling priorities. A program can modify its priority during execution. Real-time processes execute at one of the high 16 priority levels, and normal processes (including system processes) execute at one of the low 16 priority levels. The scheduler may temporarily increase the priority of a normal process to increase its response to I/O events or system events (but it can never lower the priority of a real-time process).

Batch and printer output processing is completely flexible. The operator controls the number of batch jobs that can run concurrently. The operator defines the number of spooler queues. There can be multiple print queues: a generic queue for jobs that can be output on any printer, and several queues for jobs that are designated for a specific printer.

Batch jobs can be submitted to batch streams from the interactive environment using a terminal command, from another batch job, or by any program using a system call. Submitted batch jobs are queued, and a time can be specified after which a batch job should be executed.

Flexibility in Programming Interfaces

The I/O programming facilities can be as device-independent or device-specific as desired. The record management services support high-level programming languages by providing transparent record accessing, but also enable the programmer to request specific record management services or system services to control file allocation, record blocking, or record accessing directly. Programmers can also use the system services to access file-structured devices or non-file structured devices if they wish to use their own record processing or volume structuring techniques.

Access to network facilities is device-independent, but a user who so desires can exert control over operations to obtain error reports or notification of broken connections (interrupt messages, inbound connections). System access protection applies to all network access.

Extending the System

The VAX/VMS command language can be extended with user-defined commands through the use of command procedures. A command procedure is a set of commands, data, or other command procedures processed in sequence. The user can invoke command procedures by a single command that can include parameters for the procedure, such as file names or values for symbols. Command procedures can execute programs, transfer control within the command procedure conditionally or unconditionally, request input from the user, and manipulate numeric and string symbols.

VAX/VMS uses a standard procedure call interface supported by the processor's call instructions. The calling program and called procedure can be written in different languages. This contributes to the writing of error-free, modular, and maintainable software that can be shared by many programs. The standard procedure call interface is particularly useful to systems programmers who want to add their own sharable libraries and library procedures to the VAX/VMS Common Run-time Procedure Library.

PDP-11 Compatibility

Users who already know the PDP-11 will find the native VAX-11 instruction set and programming characteristics easy to learn when developing new applications for the VAX-11/780 system. The PDP-11 and the VAX-11/780 systems have almost identical FORTRAN IV-PLUS, BASIC-PLUS-2, and COBOL languages. Users who have programmed in any of these languages on the PDP-11 will need to spend very little time learning the VAX-11/780 system.

VAX-11/780 offers many PDP-11 compatibility features:

- The VAX-11/780 processor can execute a subset of PDP-11 16-bit instructions in compatibility mode.
- The VAX/VMS operating system provides functionally equivalent system services for many RSX-11M executive directives.
- The VAX/VMS high-level language compilers accept source languages that are upward compatible with the same PDP-11 compilers.
- The VAX/VMS file system can read and write disk volumes and magnetic tapes written under RSX-11 and IAS operating systems.
- The VAX/VMS record management services provide record processing methods that are upward compatible with RMS-11 record management services.
- The VAX/VMS operating system provides an RSX-11 MCR command language interpreter.
- The DECnet/VAX package supports RSX-11S system image down-line loading.

These features make VAX-11/780 an ideal host system to PDP-11 systems in a distributed processing environment.

BLANK

3 The Users

This system is designed to execute many different kinds of jobs concurrently. Jobs consist of one or more processes, each of which can be executing a program image that interacts with on-line users, controls peripheral equipment, and communicates with other jobs in the same system or in remote computer systems. Jobs include:

- customer-written interactive, batch, and real-time applications
- interactive and batch program development jobs
- system management and control jobs

This system's general users are those people who interact with application or system jobs at an on-line terminal, or who benefit from production batch or real-time jobs. To aid in the development of interactive, batch, and real-time applications, and manage and control system resources, this system enables:

- *the application programmer* to write, compile, and test programs interactively or in batch mode, taking advantage of source code, object code, and program image libraries
- *the system programmer* to design application systems that require a high degree of job and process interaction, data sharing, response time, and system and device independence
- *the system manager* to authorize users, limit resource usage, and grant or restrict privileges individually
- *the system operator* to monitor operations, service user requests, and control batch production

People who use this computer system can control its operation directly through the operating system's command language. In general, the command language is used by programmers to develop application software, by operators to monitor the system, and by system managers to assign user privileges.

Application users themselves also employ the command language to run their application programs explicitly. The command language can be extended easily to provide specific commands for their use. Customer-written application programs can provide their own command interfaces for people using the system. Transaction processing applications can designate some terminals as slave terminals, meaning that they are tied to particular application programs that handle requests typed by the user.

The system manager can assign access user names and passwords to users who log on the system at a command terminal, and determine their privileges for obtaining services and limits for using resources. Users who access the system through an application terminal interface have the resources and privileges granted to the application programs run on their behalf. An application program itself determines who can request its services.

An operator uses the command language to control operations, check system status, and run utility programs. Operators define the number and kind of I/O spool queues and the number of batch job streams in the system. An operator can run diagnostics to check the operation of both hardware and software, and can generate system statistics reports.

THE APPLICATION PROGRAMMER

The application programmer has four basic tools for requesting services of the system:

- command interpreter
- programming languages
- programmed file and record management services
- programmed system services

The application programmer gains access to the system through the command interpreter. The command interpreter enables the programmer to create, compile, and execute programs written in any of several programming languages. The record management services are available through any programming language to provide device-independent data processing. The system services, although primarily of interest to systems programmers, are also available to the application programmer for requesting special services of the operating system.

The Command Language

The command interpreter is interactive, comprehensive, easy to use, and extremely flexible. It enables the user to log on the system, manipulate files, develop and test programs, and obtain system information. Furthermore, it enables users to extend or redefine their command repertoire easily. The command language includes:

- a set of English commands that provide the basic command repertoire
- a set of control characters that provide special functions such as erase command line, interrupt the program currently executing, etc.
- a set of special operators and commands that can be used to automate command streams and extend the command repertoire

Table 3-1 lists the basic set of English commands accepted by the command interpreter. The command interpreter is easy to use because its commands can be abbreviated, because it prompts for necessary arguments, and because it assumes standard or user-selected default values for command parameters and qualifiers.

Table 3-1
Basic Command Summary

GENERAL SESSION INFORMATION AND CONTROL

HELP	Displays information to assist the user in selecting the proper command qualifiers.
SHOW	Displays any of the following information: current day, date, and time; user's current file specification defaults; user's current session accounting and resource allocation statistics; status of user's current queue jobs; user's current default file protection; user's current terminal characteristics; symbolic name assignments; logical device name assignments; logical name translations.
SET	Changes a file's protection; changes the user's terminal characteristics; changes the user's password; changes the user's default device name or directory name; requests or cancels suppression of printing the command lines executed in a command procedure.
ASSIGN	Assigns a logical name to a given character string (equivalence name) and stores the the pair of names in a process, group, or system logical name table. Generally used to create a logical name for a device.
DEFINE	Creates a logical name equivalence. (Same as ASSIGN except for syntax.)
DEASSIGN	Breaks the correspondence between a logical name and its equivalence name (see ASSIGN and ALLOCATE), or deletes a symbol (see DEFINE).
MCR	Signifies that the given command or following command lines are to be interpreted by the RSX-11 command interpreter.
SEND	Sends the given text to an operator's terminal.
LOGOUT	Terminates an interactive session and releases all resources allocated to the user.
REQUEST	Displays a message at a system operator's terminal and optionally requests a reply.

BATCH AND COMMAND PROCEDURE SPECIFIC CONTROL*

SUBMIT	Places a given batch command file or command procedure in a batch queue for processing.
$JOB	Indicates the beginning of a batch command file and provides job control information (such as time limit).
$INQUIRE	Requests interactive assignment of a value to a symbol and assigns the symbol a name.
$GOTO	Transfers flow of control to a given labeled line.
$ON	Transfers flow of control to a given labeled line if an error of a given severity or greater is encountered at any time during command procedure processing.
$IF	Transfers flow of control to a given labeled line if the result of a logical comparison of symbolic values is true.
$DATA	Indicates that the following data in the input stream contains a $ in record position one.
$EOD	Signifies the end of data in the input stream following a $DATA command.
$EXIT	Terminates the command procedure.

*Command names preceded by a $ are meaningful only in a batch command file or command procedure. All other commands listed in this table can either be issued interactively or used in a batch command file or command procedure.

VOLUME AND DEVICE RESOURCE CONTROL

MOUNT	Requests the operator to make a volume available to the user and optionally associates a logical name with the volume or volume set.
INITIALIZE	Writes a directory file and other volume structuring information on a disk or magnetic tape volume to prepare it for use.
DISMOUNT	Requests the operator to break the logical association of this device with the user's job.

Table 3-1 (cont)
Basic Command Summary

ALLOCATE	Obtains exclusive ownership of device and enables the user to assign a logical name to the device.
DEALLOCATE	Releases allocated devices.
FILE MANIPULATION	
DIRECTORY	Reports information (size, protection, ownership, creation time, etc.) on a given file or set of files.
CREATE	Creates a new file from data subsequently entered in the input stream (user at terminal or batch stream). Creates a directory file on a volume.
EDIT	Opens a text file and accepts commands to insert, delete or modify data in the file.
DELETE	Deletes a given file or set of files.
PURGE	Deletes all but the latest version of a given file or files, optionally keeping the latest two or more versions.
RENAME	Changes the name of one or more existing files.
COPY	Copies the contents of a file or files, creating another file or files.
APPEND	Concatenates the contents of sequential files to a given output file, or creates a new output file from the concatenated contents of given sequential files.
DIFFERENCES	Compares two files and reports the differences between the two.
SORT	Creates a file by rearranging the records in a given file based on the contents of key fields within the records.
OPEN	Opens a fie for reading or writing at the command level.
CLOSE	Closes a file that was opened for input or output with the open command and deassigns the logical name specified when the file was opened.
READ	Reads a single record from a specified input file and equates the record to a specified symbol name.
WRITE	Writes a record to a specified output file.
PRINT	Sends the contents of a given file or files to a spooled output device such as a line printer.
TYPE	Displays the contents of a given file or files on the device identified by the logical name SYS$OUTPUT: (default generally the user's terminal).
DUMP	Produces a printed listing of the contents of a file, ignoring any print formatting characters that may appear in the records.
UNLOCK	Permits access to a file that was improperly closed.
PROGRAM DEVELOPMENT AND EXECUTION CONTROL	
MACRO	Assembles given assembly language source modules, producing an object module.
FORTRAN	Compiles given FORTRAN language source modules, producing an object module.
BASIC	Compiles given BASIC language source modules, producing an object module.
COBOL	Compiles given COBOL language source modules, producing an object module.
LIBRARIAN	Creates, deletes, or maintains libraries of object modules, sharable images, or macro source modules.
LINK	Links modules to produce images.
RUN	Executes a program image in the current process context, or creates a detached process and executes a program image in that process context.
DEBUG	Starts interactive debugging session after interrupting program image execution by typing a Control C or Control Y.
EXAMINE	Displays the contents of a location in virtual memory.
DEPOSIT	Replaces the contents of a location in virtual memory with the given data.
CONTINUE	Resumes execution of a program interrupted by typing a Control C or Control Y.
STOP	Terminates the program currently interrupted by a Control C or Control Y.
SUBMIT	Enters a command procedure in the batch job queue.
SYNCHRONIZE	Places the process executing a command procedure in a wait state until a specified batch job completes execution.
WAIT	Places the current process in a wait state until a specified period of time has elapsed.
CANCEL	Cancels scheduled wakeup requests for a specified process. This includes wakeups scheduled with the run command and with the schedule wakeup ($SCHDWK) system service.

A command line normally consists of a command verb followed by one or more parameters that identify the object of the operation (a file, for example) or qualify how the operation is to be performed. If the interactive user enters an incomplete command, the command interpreter prompts for any necessary parameters. For example, the COPY command, which creates a copy of an existing file, accepts a minimum of two file specifications: one for the file to be created and one for the file to be copied. The file specifications identify the exact location and name of the files. The COPY command can be entered in any of several ways:

```
$ copy
$_FROM? file-name-1
$_TO? file-name-2

$ copy file-name-1
$_TO? file-name-2

$ copy file-name-1 file-name-2
```

The command interpreter displays the dollar sign to prompt for a command, and the dollar sign underscore to prompt for a missing parameter.

To eliminate the need for typing frequently repeated sequences of commands, users can create **command procedures**. A command procedure is a file containing complete command lines (including the $ prompt character). The user can request the command interpreter to read and process the command lines in a command procedure file just as if they were being typed successively at the terminal. To execute a command procedure, the user simply precedes the name of the command procedure file with an "at" sign (@):

```
$ @procedure-file
```

Because command procedures give the user the ability to execute stored command streams, the command language itself becomes a programming language. To support command procedure programming, the command language includes commands that enable the user to label and comment command lines, and control command procedure processing flow. For example, the user can label a command line and issue the $GOTO command to transfer control to that line.

The command language also includes commands and special operators that enable the user to define symbols and assign these symbols values. For example, the $INQUIRE command requests the user to enter a value for a given symbol during command procedure processing. Special operators enable the user to assign numeric or string values to a symbol and manipulate those symbols in expressions. Symbols can be defined as local or global, that is, available only to the procedure in which they are defined or available to all procedures executed during a session. Users can set up symbolic names automatically for a command session. When a user logs on the system, the login program image executes a command procedure called LOGIN.COM automatically.

These commands and operators enable the user to extend existing commands or create new commands. Figure 3-1 is an example of how the user might create a new command called TEST. To the user, TEST appears to be a command like any other command in the command repertoire. TEST takes one parameter: the name of a FORTRAN source program to be compiled, linked, and executed. If there is an error at any stage during processing, the command procedure enables the user to decide whether to terminate or continue the command procedure.

The user creates a command procedure named TEST.COM and equates the symbolic name TEST to the command string (enclosed in double quotes) that executes the command procedure:

$ TEST := "@TEST.COM"

This command procedure compiles, links, and executes a FORTRAN program whose name is supplied as a parameter by the user executing the command procedure. The system automatically defines up to eight parameters (named P1 through P8) when a user executes a command procedure. In this example, the command procedure uses the parameter P1 for the name of the FORTRAN program the user wants to test.

For example, if the user enters the following command line, the command interpreter executes the TEST.COM command procedure and assigns the parameter P1 the name of a FORTRAN program, HANOI, to be compiled, linked and executed:

$ TEST HANOI

The command procedure loops through each stage of processing using a symbol named STEP as a step counter. Note that where symbols appear within command lines as values to be interpreted, they are normally enclosed in single quotes. Where they appear at the end of the line, the closing quote is not necessary.

If an error occurs during any stage of processing, the command procedure executes the routine beginning at the line labelled CHECK. The routine requests the user to enter a value for the symbol CONFIRM. The IF statement tests the value of CONFIRM, taking advantage of the fact that the system automatically gives a symbol the value true if the user enters 1 (for TRUE) or if the first character of a string is a Y (for YES). If an error occurs (such as a warning during compilation that the user chooses to ignore), the user can enter any string beginning with the character Y and the routine will reset the ON WARNING THEN GOTO CHECK statement and continue with the next stage of processing.

```
$ ON WARNING THEN GOTO CHECK  ! If any warning, error,
                              ! severe error is detected
                              ! during compilation, then
                              ! transfer control to the line
                              ! labelled CHECK.

$ STEP = 0                    ! Initialize the value of the
                              ! symbol named STEP to 0.

$ LOOP: STEP = STEP + 1       ! Increment STEP to set up each
                              ! stage of processing.

$ IF STEP.LE.3 GOTO 'STEP     ! Depending on the value of
                              ! STEP, go to the appropriate
                              ! line labelled with that value.

$ EXIT                        ! (Otherwise) Terminate the
                              ! command procedure.

$ 1: FORTRAN 'P1              ! Compile the FORTRAN program
                              ! whose name is suppled as the
                              ! value of P1.

$ GOTO LOOP                   ! If there are no errors,
                              ! continue with the next step.

$ 2: LINK 'P1                 ! Link the program
                              ! just compiled.

$ GOTO LOOP                   ! If there are no errors,
                              ! continue with the next step.

$ 3: RUN 'P1                  ! Run the program just
                              ! linked.

$ GOTO LOOP                   ! Complete the loop.

$ CHECK: $INQUIRE CONFIRM     ! Request the user to enter
                              ! a value for CONFIRM.

$ IF .NOT.CONFIRM THEN EXIT   ! Terminate processing if the
                              ! user entered anything other
                              ! than a string beginning with Y.

$ ON WARNING THEN GOTO CHECK  ! Otherwise, reset error check.

$ GOTO LOOP                   ! And continue processing.
```

Figure 3-1
Extending the Command Language

In addition to executing command procedures at a terminal, an interactive user can also submit batch jobs. Batch jobs execute under control of the system operator and leave the user's terminal free to continue interactive or command procedure processing. A batch job can be submitted as a deck of cards or as a batch command file. A batch command file is identical to a command procedure file, except that a batch command file submitted as a deck of cards begins with a $JOB card that provides job control information.

The RUN Command

The RUN command includes several qualifiers (DELAY, INTERVAL, and SCHEDULE) which are of particular importance to the real-time programmer.

Specifying any of the above qualifiers places a process in hibernation, a wait state in which the process can be reactivated only when a particular time value occurs. The time value can be specified in delta time (/DELAY qualifier), in absolute time (/SCHEDULE qualifier) or at recurrent intervals (/INTERVAL qualifier). When the image completes execution, the process returns to a state of hibernation.

The Programming Languages

The system includes the VAX-11 MACRO assembler for programming the machine using its native instruction set. Five language processors are optionally available to high-level language programmers: FORTRAN, COBOL, BASIC, RPG II and BLISS. These language processors, introduced below, are described further in the section on Languages.

The FORTRAN language processor, VAX-11 FORTRAN IV-PLUS, is an extension of the PDP-11 FORTRAN IV-PLUS language processor. The FORTRAN IV-PLUS language is based on the 1966 American National Standard FORTRAN specification, and provides many useful extensions to the standard.

The VAX-11 FORTRAN IV-PLUS language processor produces native mode code that takes advantage of the system's large virtual address space, floating point instructions, and character instructions. FORTRAN I/O processing is supported by the record management services. FORTRAN programs can call the record management services explicitly, as well as user-written assembly language routines and the system services. FORTRAN IV-PLUS object modules can be linked with assembler-produced object modules and the system's run-time procedure library, which is common to all native mode programs, to provide standard library functions.

The COBOL language processor, VAX-11 COBOL-74, accepts a source language highly compatible with the PDP-11 COBOL source language; it produces a highly efficient native mode code which utilizes the system's packed decimal and character instruction set and extended call facility. The VAX-11 COBOL-74 language is based on the 1974 American National Standard COBOL low-level specification, and supports the high-level specification in the Nucleus, Table Handling, Sequential I/O, Relative I/O, Indexed I/O, and Segmentation modules.

The BASIC language processor, PDP-11 BASIC-PLUS-2/VAX, produces compatibility mode code. It accepts a source language identical to the PDP-11 BASIC-PLUS-2 source language, with the addition of VAX/VMS file specifications. BASIC-PLUS-2 is an extension to the BASIC language, which was developed at Dartmouth College as one of the simplest programming languages to learn. The BASIC-PLUS-2 extensions to BASIC provide experienced programmers with advanced data manipulation and procedure expression techniques.

PDP-11 RPG II/VAX is an optional language processing system that accepts a source language identical to the PDP-11 RPG II source language, with the addition of VAX/VMS file specifications. It is designed specifically for producing printed reports. The language is characterized by simple-to-use source specifications and a limited command structure. Several utilities are also available which simplify coding of specifications and assist in converting source programs and tapes.

BLISS-32 is a high-level systems implementation language for VAX-11, which runs in native mode under VAX/VMS. It is characterized by a unique blending of high-level and low-level language elements and also contains structures not found in any existing language. BLISS-32 is designed to be executed on architecturally different machines with little or no modification. BLISS-32 can be used as an alternative to assembly language coding in all except the most machine-dependent systems programming applications.

Record Management Services

The record management services (RMS) are a collection of procedures that extend the programming languages by providing general purpose file and record handling capabilities. Programmers using RMS include in their programs statements that read, write, find, delete, and update records within files. Records can be fixed or variable length.

RMS enables the programmer to choose the file organization and record access method appropriate for the data processing application. The file organizations and record access methods are independent of the language in which they are programmed, although some languages support file organizations and access methods not provided in others. Every programming language uses RMS to process files organized to provide sequential or random record accessing. In addition, the BASIC-PLUS-2 and COBOL languages can be used to process multikey indexed files sequentially or randomly by index key field values.

For further information on RMS and the system's data management techniques, refer to the section on Data Management Services.

THE SYSTEM PROGRAMMER

The system programmer can use this system to design and build application systems for multiprogramming environments requiring fast response and a high degree of job interaction and data sharing.

Job and Process Structure

The user program environment consists of a job structure that can contain many processes. A process is the schedulable entity capable of performing computations in parallel with other processes. It consists of an address space and an execution state that define the context in which a program image executes. An executing program is associated with at least one process, but it can be associated with several processes.

A multiple process job structure allows one job to execute more than one program image at the same time. One process can wait for an event (such as I/O completion) to occur while another process continues its computations. The processes can communicate in several ways. They can coordinate their execution synchronously using event flags or asynchronously using software-simulated interrupts. They can send messages back and forth using virtual record-oriented devices called "mailboxes," and they can share code and data on disk and in memory.

Jobs can be grouped into application subsystems that share code and data protected from other applications. The processes within jobs in the same group can coordinate their activities using group interprocess communication facilities such as mailboxes and event flags, as well as those facilities local to the job. They can access files and data in memory that are protected from other groups in the system.

Multiprogramming Environment

The system supports multiprogrammed applications that require high performance by providing:

- event driven priority scheduling
- rapid process context switching
- minimum system service call overhead
- processor access mode memory protection
- memory management control

The system schedules processes for execution based on the occurrence of events such as I/O completion rather than time quantum expiration. When scheduled, the context switching and interrupt processing hardware and software ensure that processes are activated quickly. Real-time processes can be assigned high priorities to ensure that they receive processor time on demand. A process can schedule its execution at a given time of day or after an interval has elapsed, and an appropriately privileged process can modify its priority during execution.

The system's memory management hardware and software ensure that paging, swapping, and dynamic memory allocation are both efficient and transparent to the programmer. Where real-time applications require performance control, both paging and swapping can be reduced or eliminated by increasing the amount of memory allocated to a process and by locking a process in memory. Because memory management is transparent, programs can be written and later tuned for performance after they are tested. The system provides a utility program to aid system programmers in evaluating the effectiveness of the memory management system for their processes.

Program Development

This system provides the system programmer with tools that support highly modular program and applications development. By taking advantage of these tools, the programmer can build applications quickly, and easily modify and extend them later.

The system includes editors, compilers, librarians, linkers, and debuggers for both the native programming environment and the compatibility mode programming environment. All program development utilities can be used either interactively or in batch mode, including the editors and debuggers. The native symbolic debugger recognizes a command language similar to the operating system command language and uses expressions similar to the language in which the program being debugged was written.

Executable program images can be built using extensive libraries. In the native programming environment, the programmer can create libraries of assembler macro definitions, of object modules, and of image modules. The system also includes the common run-time procedure library, which provides library functions common to all native programming languages.

All program interfaces to the operating system and its utilities have uniform calling standards. System programmers can add new library procedures to the common run-time procedure library and install them on-line without modifying existing programs and utilities, since all arguments are passed using standard data structures.

Furthermore, user programs can be written to be completely device independent through the system service and command language logical naming facilities. All files and devices can be identified using arbitrarily defined logical names that can be assigned values at run time.

APPLICATIONS EXAMPLES

To illustrate how the multiprogramming capabilities of the system can be effective in widely diverse applications, Figures 3-2 and 3-3 show two hypothetical application systems: a commercially oriented data processing system and a real-time flight training simulation system.

Commercial System Example

The commercial system diagram (Figure 3-3) begins with both a programming group and a data processing operations group. Within the programming group, jobs can be performing requests for programmers at terminals who are using the system's text editors, compilers, and linkers to write and test programs for both the VAX-11/780 and an RSX-11M system in the manufacturing department. The programmers can execute command procedures and submit batch jobs to automate repetitive development steps.

Within the operations group, the system's operators can be managing batch and spool queues, backing up disks, and monitoring performance. They may be down-line loading tasks into the RSX-11M system. They may be running an accounting program that interprets and summarizes the accounting statistics collected by the operating system during the period and sends that data to the business data processing subsystem.

A billing process within the business data processing group can be suspended until it is activated by a process, such as the accounting process, that wants to send it billing information. The accounting process can send to the billing process the name of an account summary file that it created, using a permanent mailbox defined for that purpose.

The business data processing group may also run a job that handles order entry terminals. The job can consist of a controlling process that handles input from the terminals,

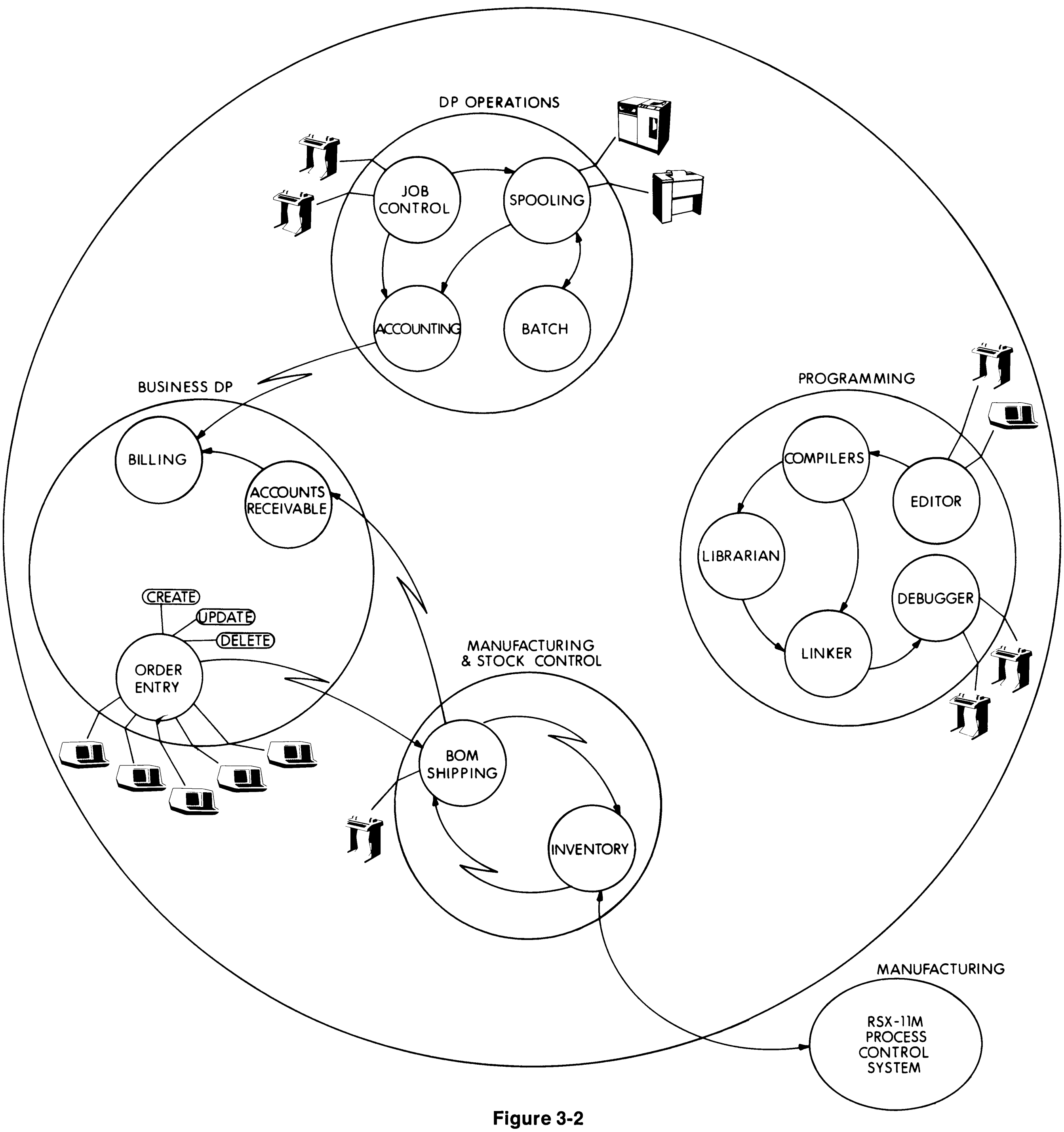


Figure 3-2
Application Subsystems

and several subprocesses that perform the actual record processing functions. As users at the terminals enter their requests, such as create order record, update order record, etc., the controlling process collects the input from a terminal and sends the request to the appropriate subprocess's mailbox. The subprocess may be hibernating, having requested the operating system to activate it when anything is written to its mailbox. Or the controlling process can simply set an event flag to notify a subprocess that it has received a request for which the subprocess is waiting.

A process in the manufacturing application group may regularly collect orders from the order entry job to create materials parts lists, or notify stockroom clerks of high-priority orders. The stockroom clerks can keep inventory records up to date by using the shipping and inventory jobs, because they can collect manufacturing statistics from a background task in the RSX-11M process control system on the manufacturing floor. Once orders are shipped, the shipping process can notify an accounts receivable process in the business data processing application group, which in turn can activate the billing process.

VAX-11/780 SAMPLE APPLICATION SYSTEM
FLIGHT TRAINING SIMULATION

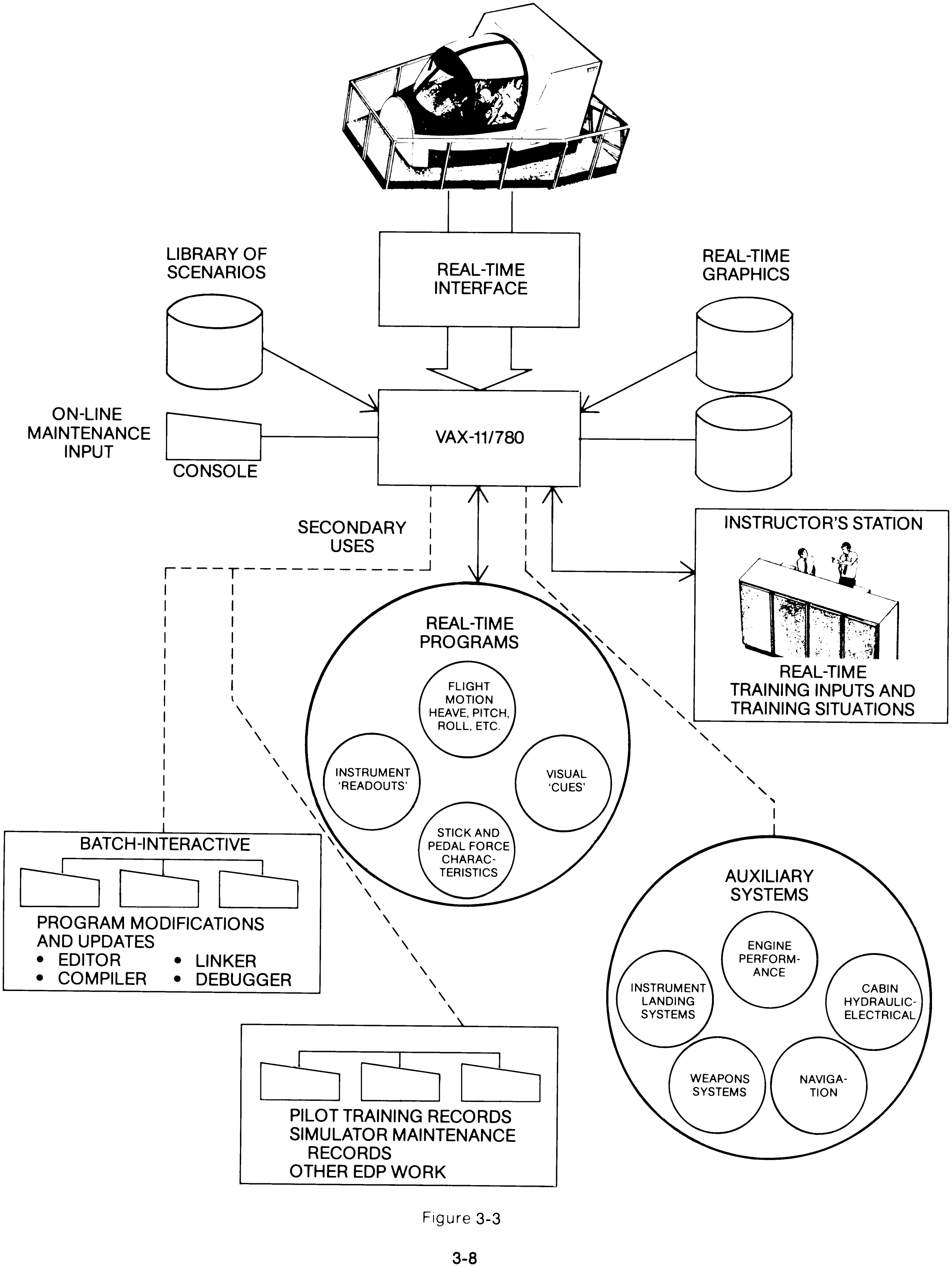


Figure 3-3

REAL-TIME FLIGHT SIMULATION EXAMPLE

To illustrate how VAX's facilities can be as extended to the real-time environment, Figure 3-3 shows a sample flight training simulation system. Flight simulation is a particularly good application for the VAX-11/780, since in addition to fast real-time response, such systems must also be capable of solving large and complex equation systems. In addition, such systems also require general program development facilities such as FORTRAN compilation, assembly, editing, debug, library facilities, and fast-access file management.

The illustrated system shows how the multiprogramming capabilities of the VAX-11/780 allow it to handle the basic real-time tasks of data acquisition and transmission, while also performing a wide range of other activities:

(Looking from top to bottom) the diagram begins with a representation of an aircraft fuselage containing the cockpit throttle levers and control panel which the student operates to simulate flight. The signals and control movements of the student go through an analog to digital conversion, and are then passed through a real-time interface device into VAX memory. Before the system can respond to the data generated by the student, complex flight-motion equations must be called and combined with current aircraft data. This, in turn, produces a new set of circumstances with which the student must deal.

Additional inputs to the system are also provided by an instructor at a timesharing terminal (shown in the right central part of the diagram). By typing commands at the terminal, the instructor can control the situations which the student must face — for example, by injecting an engine failure, weather change, or some other complication. The instructor can also introduce additional variables into the system by inputting pre-defined "scenarios" which have been stored on libraries. The student's flight environment can include "visual cues" (e.g., terrain, runway approaches, other aircraft, etc.) produced by sophisticated, real-time graphics modules.

In addition to performing basic flight simulation functions, the system also performs a number of auxiliary functions which are less time-critical in nature. These would include such activities as monitoring of engine performance, testing of navigation and instrument landing systems, testing of weapons systems, and monitoring cabin, hydraulic, and electrical systems. The system also generates a file of student performance statistics which can be analyzed at a later time.

The system also provides facilities for program development. As shown in the lower left-hand side of the diagram, programmers at terminals can write and test new applications programs (in either batch or interactive mode) using text editors, compilers, linkers, and debuggers. Because of the way the VAX-11/780 architecture handles real-time vs. normal processes (described below), program development can take place simultaneously with the running of the flight simulator; no significant reduction in real-time processing speed will result.

Design Considerations

Systems such as that shown in Figure 3-3 must be designed to operate at extremely high speeds, both in terms of real-time I/O and computation. Many simulators require turnaround times (data sampling, computation, and output) in the 25-50 millisecond range.

There are several elements of the VAX hardware and VAX/VMS operating system which aid in achieving such speeds. Most important is VAX's context switching ability — a result of special processor instructions which relieve the operating system software of having to individually load or save the hardware registers which define the hardware context. Another element is the design and operation of VAX/VMS device drivers. VAX/VMS drivers are forked processes which are created dynamically in response to a user I/O request or unsolicited device interrupt. They operate with minimal context, execute to completion when invoked, and remain memory resident throughout execution. (VAX/VMS device drivers can be written for user-developed devices which attach to the VAX-11/780 UNIBUS. The manual entitled *VAX/VMS Guide to Writing an I/O Driver* describes how to write, test and load them.)

In addition, system designers can utilize the VAX/VMS scheduling and system service facilities to yield still higher processing speeds. For example, the following design schema could be employed:

Those processes which perform the basic flight simulation functions (i.e., flight motion equations, visual graphics modules, etc.) can be designed as a group of hibernating processes capable of being immmediately reawakened as required by the student's control activities. This could be accomplished via the use of system service calls such as ($HIBERNATE/$WAKE) or ($SUSPEND/$RESUME) or by using interprocess control structures such as mailboxes and common event blocks.

To further ensure that basic simulation functions will operate at the fastest possible speed, real-time processes can be granted the highest scheduling priorities. Such processes can also be given special privileges which allow them to eliminate paging and swapping and thus assure memory residency. (Note that when a VAX process runs at real-time priority its priority level will actually be higher than system processes such as the Pager, Swapper, Linker, and Scheduler itself).

Processes which are not time-critical can be assigned normal scheduling priorities. Those processes which perform what is essentially a monitoring function (e.g., engine performance, electrical system, etc.) can also be hibernated, then monitored periodically via the issuance of a programmed system timer service. Other activities such as program development and statistical processing can take full advantage of the VAX/VMS system with minimum impact upon the basic real-time core of the simulation system.

For more information on the concepts of jobs, processes and program images, refer to the section on the Operating System.

THE SYSTEM MANAGER

A job is normally associated with a user known to the system to have certain privileges, quotas, and resources. The system manager authorizes users, plans data access and protection, grants privileges, controls resource utilization, and analyzes the system's accounting and performance information.

User Authorization

The system manager controls use of the system primarily by creating user authorization information. This information is recorded in a specially maintained and protected file called the **user authorization file**. The system manager can create, examine, and update this file at any time.

The file contains one entry for each user authorized to access the system. Each entry:

- identifies the user
- supplies defaults
- specifies privileges
- limits resource usage

User identification consists of a unique user name, a password, a default account name, and a user identification code (UIC). When logging on the system, people must always enter their user name and password to gain access to the system. The password is not displayed on the user terminal. Users can change the passwords they are assigned as often as they desire.

The user logging on to the system can also enter an account name under which the system's accounting statistics are to be collected for the given session. Account names are arbitrary character strings created by the system manager. If the user does not enter an account name, the system uses the default account name appearing in the user authorization file.

This system's data protection scheme is based on the user identification code (UIC) that the system manager assigns. A UIC controls each user's access to the data structures protected by UICs, which include both files and the interprocess communication facilities such as mailboxes, shared areas of memory, and event flags.

A UIC consists of a group number and a member number. Every user is assigned a UIC, and every data structure is assigned both a UIC and a protection code. A protection code identifies what types of access are available to which users. There are four types of access (read, write, execute, and delete), and there are four types of user (owner, group, world, and system). The owner is any user that has the same UIC as that assigned the data, the group is any user that has the same group number as that assigned the data, the world is any user, and the system is any user with a group number of 1 through 8.

Using this protection scheme, a system can have files and interprocess communication facilities that are available for access only by users having the same UIC, or for access only by users in the same group, or for universal access. Futhermore, since each data structure has its own protection code, it is possible to protect each data structure assigned the same UIC on a different basis. The system UICs are generally reserved for system users and system programs and data structures. This arrangement enables a user to protect a file from access by anyone other than the owner or group, but still enables the system to access the file for operations such as backup.

In addition to identifying the user and the set of data structures the user can access, the user authorization file supplies the user with a default file protection, a default directory name, and a default device name. When the user creates a file, the system assigns the default file protection unless requested otherwise. An owner can modify a file's protection at any time.

Directory names are arbitrary character strings identifying a directory file. A directory is simply a file containing a list of file names and other identification information that is used to find files on a volume. The default directory name identifies the directory that lists the files the user normally accesses. The default device name is the name of the device on which the volume containing the files the user normally accesses is mounted.

When the user issues a command to the command interpreter that operates on a file, or runs a program that opens a file, the file system uses the default directory name and default device name to locate the file unless specifically requested to use some other directory name or some other device name. The user can change the default directory and device names for a given session.

For further information on directories and directory structures, refer to the section on Data Management Services.

Privileges

Each user's authorization file entry contains a list of the privileges that the user can invoke. They include interprocess communication and control privileges, performance control privileges, file and device access privileges, and system operational control privileges. The system manager can grant distinct privileges individually to each user. Table 3-2 lists some of the privileges.

Table 3-2
Privileges Summary

INTERPROCESS CONTROL

- Create event flag clusters
- Create permanent common event flag clusters
- Create temporary mailboxes
- Create permanent mailboxes
- Create global sections
- Suspend, resume, wake, and delete processes within the same group
- Suspend, resume, wake, and delete any process
- Create detached processes

ACCESS TO FILES AND DEVICES

- Insert logical names in group logical name table
- Insert logical names in system logical name table
- Allocate spooled devices
- Obtain exclusive ownership of a shared device

PERFORMANCE CONTROL

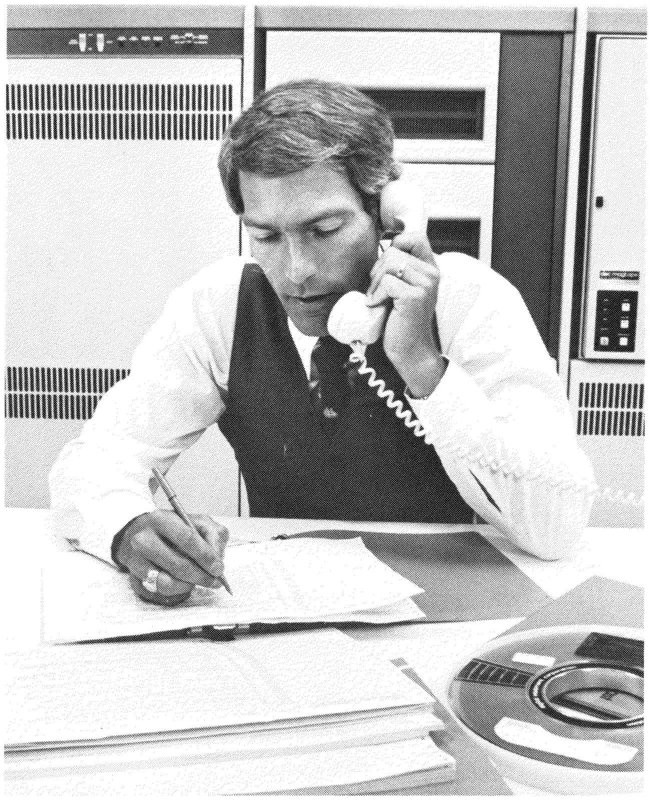
- Execute time critical images
- Lock process in memory

SYSTEM OPERATION CONTROL

- Issue operator commands
- Set any privilege bits
- Set process priority

PROGRAM EXECUTION

- Execute Change Mode to Executive system service
- Execute Change Mode to Kernel system service
- Break through UIC protection
- Bypass file protection
- Break through device ownership of another process
- Change the characteristics of a device
- Issue diagnostic functions
- Send messages to error logger
- Execute "pass all characters" terminal read requests
- Suppress accounting messages



Privileges are checked when the user executes program images. If a user runs an image that attempts to execute a function requiring a privilege the user is not granted, the image incurs a privilege violation. For example, diagnostic programs require the privilege to issue device level diagnostic functions and the privilege to send messages to the error logger. Users not granted these privileges will receive privilege violations if they attempt to run diagnostics.

In certain cases, however, it is desirable to let a user run an image that requires privileges the user is not granted. For example, the login program image requires the privilege to switch to a more protected processor access mode to set the user's initial context in a protected area of memory. To let a user run an image that requires special privileges, the system enables the system manager to install **known images**. When the user runs a known image, the user obtains the necessary privileges to execute the functions required by the image, but only for the duration of that image's execution.

Resource Quotas and Limits

The user authorization file also provides the limits on how many system resources a user can tie up while logged on the system, and quotas for how much of a resource a user can use up during an accounting period. The system manager can give users quotas for the maximum amount of CPU time they can accumulate during a given accounting period. The system manager can limit the amount of dynamic system memory a job can tie up for buffers, or the amount of disk space a job can use for paging out modified pages (see the Operating System section). Limits include the maximum number of:

- outstanding open files
- outstanding subprocesses created
- pages in a process working set
- pages in system paging files
- outstanding entries in the timer queue
- outstanding system buffered I/O requests
- bytes in system buffered I/O request
- outstanding direct I/O requests

Resource Accounting Statistics

The system maintains an accounting information file for collecting cumulative resource usage statistics. The system updates the accounting information file with detail records each time a process terminates. The detail statistics include:

- elapsed CPU time
- login (connect) time
- number of volumes mounted
- number of pages printed
- largest process virtual size
- largest process working set size
- number of page faults
- number of system buffered I/O requests
- number of direct I/O requests

A detail record identifies the account name, user name, and user identification code (UIC) to which the statistic applies. The accounting information file can be used to calculate billing information and reporting by account name, user name, or UIC. Because the system collects all detail records, system managers can define their own algorithms for resource usage billing.

Performance Analysis Statistics

The system collects statistics on its activities to help system programmers and managers tune the system for maximum performance. The information collected includes:

- System and Job Statistics — indicate the current number of processes, interactive users, and batch jobs in the system, the date and time at which the system was booted, and the current date and time.
- Processor Access Mode Usage — indicates how much time is spent executing at each of the access modes as a measure of the type of code being executed and the computational workload.
- Page Fault Activity — indicates how many and what kind of page faults occurred as a measure of the effectiveness of memory management.
- I/O Activity — indicates how many and what kind of I/O operations are taking place.
- Network Activity — indicates network workload (current number of nodes in the network, number of bytes transmitted and received, number of messages transmitted and received, number of buffers currently in use, number of successful and failing attempts to obtain space for network buffers).
- Response Time Histograms — indicate the time it takes the system to initiate user requests.

The Display Utility Program

The Display Utility Program (DISPLAY) provides a dynamic display of system performance measurement statistics on a DECscope VT52 video display terminal or on a DECscope VT55 display terminal. By typing appropriate DCL commands, system users may list information regarding I/O system activity, paging, CPU usage, current process activity, and other relevant statistics. Figure 3-4 shows a typical screen display.

Figure 3-4
Display Utility Program (I/O System Rates)

```
FREE LIST: 2016                        I/O SYSTEM RATES                                   MODIFY LIST: 35
                                           16:17:09

                              RATE     AVG                                       RATE      AVG
      NAME           VALUE    /SEC     RATE      NAME                VALUE       /SEC      RATE

DIRECT I/Os             32    7.30     1.50   PAGE FAULTS               65      14.84      1.83

BUFFERED I/Os           29    6.62     3.24   PAGES READ                 4       0.91      0.11

MAILBOX WRITES           0    0.00     0.00   READ I/Os                  2       0.45      0.07

WINDOW TURNS             3    0.68     0.14   PAGES WRITTEN              0       0.00      0.00

LOGNAME TRANS           39    8.90     0.98   WRITE I/Os                 0       0.00      0.00

FILE OPENS               3    0.68     0.07   TOTAL INSWAPS              0       0.00      0.00
```

THE SYSTEM OPERATOR

An operator is any user given the privileges by the system manager to perform operator functions. A system does not require an operator, but it can have one or several operators, and they can use any terminal to issue commands or run programs. Operator functions include:

- system startup and shutdown
- job control (change process priorities, kill jobs, etc.)
- device allocation
- volume mount and dismount request servicing
- on-line disk and magnetic tape volume and file backup
- spool and batch queue control
- software maintenance update installation
- diagnostic execution

An operator uses the command language to control operations, check system status, and run utility programs. A special system program, the Operator Communications Manager (OPCOM), is the primary operator aid. It collects and delivers the messages all users and user programs send to the operators. Any operator can respond to user requests, and the Operations Communications Manager will remind operators of any outstanding requests.

Spooling and Queue Control

The operators define the number and kind of input and output spool queues in the system. The operators can create output spool queues for any number of devices, including line printers, terminals, or even magnetic tape. The operators can also create input queues for spooling batch input from a card reader.

The operators can assign each queue a priority, merge or redirect queues to other devices, and modify the queue set-up at any time. It is possible to have more than one print queue for the same printer. For example, an operator can create a generic printer queue that will collect jobs that can be printed on any of a set of printers, and at the same time have a print queue for each individual printer. A user can issue a print request for a generic printer or a particular printer, and the operator can override the user's request.

A print job can contain one or more files to be printed together. Print jobs can be submitted by an interactive user, batch job, or any program. Print jobs are also automatically submitted at the end of a batch job. A print job can specify the forms type required, the number of copies of the job, the job priority, and a "hold until given time" request. Each file within a print job has its own copies count, and each can have these options: double space, inhibit form feed, print a flag page, label each page, or delete after printing. The operator can choose whether or not to print burst (job separator) pages, and can put jobs on indefinite hold, modify the priority of a job, or abort a job.

Batch Processing

The system supports multiple stream, multiple queue batch processing. The operators control how many batch job streams can run concurrently. Batch jobs can be submitted by an interactive user, another batch job, or any program. When the number of batch jobs submitted exceeds the number of streams, the remainder of the batch jobs are held in a batch input queue. As with the spool queues, the operators can control the batch job queue. They can change job priority, hold a job until after a given time, hold a job indefinitely, and kill a job.

Volume mount commands issued in a batch job can request a generic device, such as any disk, or a specific device, such as disk drive unit 2. The batch job waits until an operator satisfies the mount request, while other batch jobs proceed. Operators can find out which job has a given device.

On-line Software Maintenance

An operator can incorporate maintenance updates to the software without bringing down the system for stand-alone use. Maintenance patches are distributed on floppy disk and the operator simply loads the console floppy disk drive with the maintenance floppy to update the software on disk. Depending on the nature of the update, an operator may have to restart the system to activate the patched modules.

System Recovery

An operator can select manual or automatic system recovery following a power interruption or hardware or software failure.

On automatic system recovery after power interruption, the system determines whether the contents of memory are still valid, and if so, restarts all possible I/O in progress at the time of the power interruption and continues operations from the point of interruption. If the contents of memory are not valid, either because memory battery backup is not included in the configuration, or because the power failure lasted longer than the battery, the system automatically boots itself from disk and executes the start-up command procedures.

Error Logging and Reporting

The error logger is a job that the operators can initiate to run continuously during operations. It collects errors detected by both hardware and software, including:

- device errors
- interrupt timeouts
- interrupts received from non-existent devices
- memory, translation buffer, and cache parity errors

In addition, system software sends complete recovery information to the error logger following a power interruption or hardware or software failure.

The error logger writes all messages it receives into an error log file, noting vital system statistics at the time of the message. The error logger also notes benign events when they occur, such as when volumes are mounted and dismounted, and periodic time stamps indicating that no entries have occurred for a specified period of time. The error logger can accept messages from the operators at any time, and from any programs privileged to send messages to the error logger.

The system includes an error report generating program that converts the information in the log file into a text file that can be printed for later study.

On-line Diagnostics

An operator can run diagnostics to check the operation of both hardware and software. An operator can run system exercisers and device verification diagnostics while normal operations proceed. System exercisers test general purpose software and compare the results with known answers, reporting any discrepancies to the error logger.

Operators can run device verification diagnostics either stand-alone or concurrent with other processes. Diagnostics check the peripheral functions, including disk head alignment. In addition, fault isolation diagnostics, which isolate problems to replaceable units, are available for stand-alone use.

Remote Diagnosis

If the system is equipped with the remote diagnosis option, an operator sets up the system for remote preventive maintenance or troubleshooting. When a hardware error is detected or suspected, the operator mounts a diagnostic disk pack, loads a diagnostic floppy disk in the console, sets a switch on the processor console, and calls the local DIGITAL service office. An operator does not need to be present at the installation once the call is made. The DIGITAL Diagnostic Center can then connect to the installation, run automated diagnostics, operate the diagnostic console manually, and check the error log file. If a problem is found, the field service engineer can bring the proper equipment and replacement modules to make the repairs.

THE USER ENVIRONMENT TEST PACKAGE

The User Environment Test Package (UETP) consists of a series of tests designed to demonstrate that the hardware and software components of a system are in working order. The UETP consists of six phases:

- The Initialization Phase — This phase verifies that I/O devices are operational via simple read/write operations. In addition, users are prompted to supply several parameters which define the scope of the test, e.g., number of users to be simulated by UETP, amount of information to be displayed at the console, and number of consecutive runs to be made by the UETP.
- The I/O Device Test Phase — In this phase, I/O devices undergo comprehensive testing. Terminals and line printers generate pages or screens of output containing header information and a test pattern of ASCII characters. Disks and magnetic tapes are also exercised. Files are created on the mounted volumes and written to. The test then checks the written data for errors and erases the file.
- The Native Mode Phase — This phase includes three tests, each of which exercises software services provided explicitly for VAX/VMS. The first exercises VAX/VMS system services; the second exercises Native Mode Utilities such as the VAX-11 Symbolic Debugger and Image File Patch Utility; the third exercises VAX-11/RMS.
- The System Load Test Phase — This phase creates a number of detached processes which simulate the action of a group of users concurrently issuing commands from terminals; it tests the system's ability to handle various levels of utilization.
- The Compatibility Mode Test Phase — This phase tests most RSX-11M utilities running in compatibility mode on VAX/VMS.

- Termination Phase — In this phase, temporary files are deleted and other cleanup activities are performed. If multiple runs were requested during the initialization phase, then the UETP is restarted and control is passed directly to the device test phase.

The UETP is invoked via command procedures. The entire package may be specified by executing the master procedure, UETP.COM, or tests may be executed individually by specifying particular command procedures (shown in Figure 3-5).

Figure 3-5
Command Procedures for UETP

```
$ RUN  UETINIT00          Initialization Phase
$ RUN  UETINIT01
$ RUN  UNETPDEV01         I/O Device Test Phase
$ @UETCOMP00              Compatibility Mode Test Phase
$ RUN  UETNATV01
$ @UETNATV02              Native Mode Test Phase
$ @UETNRMS00
$ RUN  UETLOAD01          System Load Test Phase
$ RUN  UETTERM01          Termination Phase
```

4 The Processor

The VAX-11/780 processor is a high-speed, microprogrammed computer that executes variable-length instructions in native mode, and non-privileged PDP-11 instructions in compatibility mode. The VAX-11/780 processor includes an 8K byte cache, integral memory management, sixteen 32-bit registers, 32 interrupt priority levels, an intelligent console, a programmable real-time clock, and a time-of-day and date clock.

The VAX-11 native instruction set provides 32-bit addressing enabling the processor to address up to 4 billion (10^9) bytes of virtual address space. The processor's memory management hardware includes mapping registers used by the operating system, page protection by access mode, and an address translation buffer that eliminates extra memory accesses during virtual-to-physical address translation. Up to 8 million bytes of MOS ECC memory can be included in the system, with optional battery backup.

The VAX-11/780 provides sixteen 32-bit general registers that can be used for temporary storage, as accumulators, index registers, and base registers. Four registers have special significance: the Program Counter and three registers that are used to provide an extensive routine CALL facility. The processor offers a variety of addressing modes that use the general registers to identify instruction operand locations, including an indexed addressing mode that provides a true post-indexing capability.

The native instruction set is highly bit efficient. It includes integral decimal, character string, and floating point instructions, as well as integer, logical, and bit field instructions. Instructions and data are variable length and can start any arbitrary byte boundary or, in the case of bit fields, at any arbitrary bit in memory. Floating point instruction execution can be enhanced by an optional floating point accelerator.

The I/O subsystem consists of the processor's internal synchronous bus and the UNIBUS and MASSBUS device interfaces. The synchronous bus has a 30-bit address space and can transfer 32 bits (plus parity) or 64 bits (plus parity) every cycle. The bus is capable of an aggregate throughput rate of 13.3 million bytes per second. Parity and error checking occurs during every 200-nanosecond cycle to provide data integrity.

PROCESSOR COMPONENTS SUMMARY

The processor is the hardware logic that performs the operations requested of the computer system. Its integrated physical components are:

- the Central Processing Unit (CPU) itself, including its cache, writable diagnostic control store, optional floating point accelerator, clocks, and the console
- main memory and main memory controllers
- peripheral bus adaptors

These components communicate over a high-speed internal bus called the SBI (Synchronous Backplane Interconnect) bus. Figure 4-1 illustrates the major processor components.

The Central Processing Unit performs the logical and arithmetic operations requested of the computer system. Its user programmable registers include sixteen 32-bit general purpose registers for data manipulation, and the Processor Status Longword for controlling the execution states of the CPU. The processor's instruction set is defined by the microcode loaded in its control store.

The processor includes 12K bytes of writable diagnostic control store for updating the instruction set microcode. The writable diagnostic control store is also used for executing microcode diagnostics, which can be loaded from the console's floppy disk.

The console enables the computer system operator to control the processor operation directly. The console actually consists of an LSI-11 microcomputer and 24K bytes of memory, a floppy disk system, and a terminal. A serial line interface is optionally available for remote diagnosis.

Two memory controllers can be connected to the SBI bus. Each controller handles up to 4096K bytes of semiconductor memory, for a system total of 8192K bytes of memory. The memory controllers employ an error detecting and correcting technique that ensures single error bit correction and double error bit detection.

Also on the SBI are two kinds of peripheral bus adaptors: an adaptor for the MASSBUS, which connects high-speed disk and magnetic tape devices to the processor, and an adaptor for the UNIBUS, which connects lower-speed devices to the processor, including disks, communications lines, real-time interfaces and I/O peripherals such as terminals, line printers, card readers, and real-time DMA (Direct Memory Access) devices.

The following sections discuss the programming characteristics of the processing system from the general user's and the operating system's viewpoint. The last section discusses the processor's physical components and their features.

PROCESSING CONCEPTS FOR USER PROGRAMMING

A program is a stream of instructions and data that a user can request the operating system to translate, link, and execute. An executable program is called an **image** to distinguish it from source and object programs. When a user runs an image, the context in which the image is executed is called **a process**. A process is the complete unit of execution in this computer system. Process context includes the state of the image it is currently executing and it includes the limitations on what an image is allowed to do, which primarily depend on the privileges of the user who runs the image.

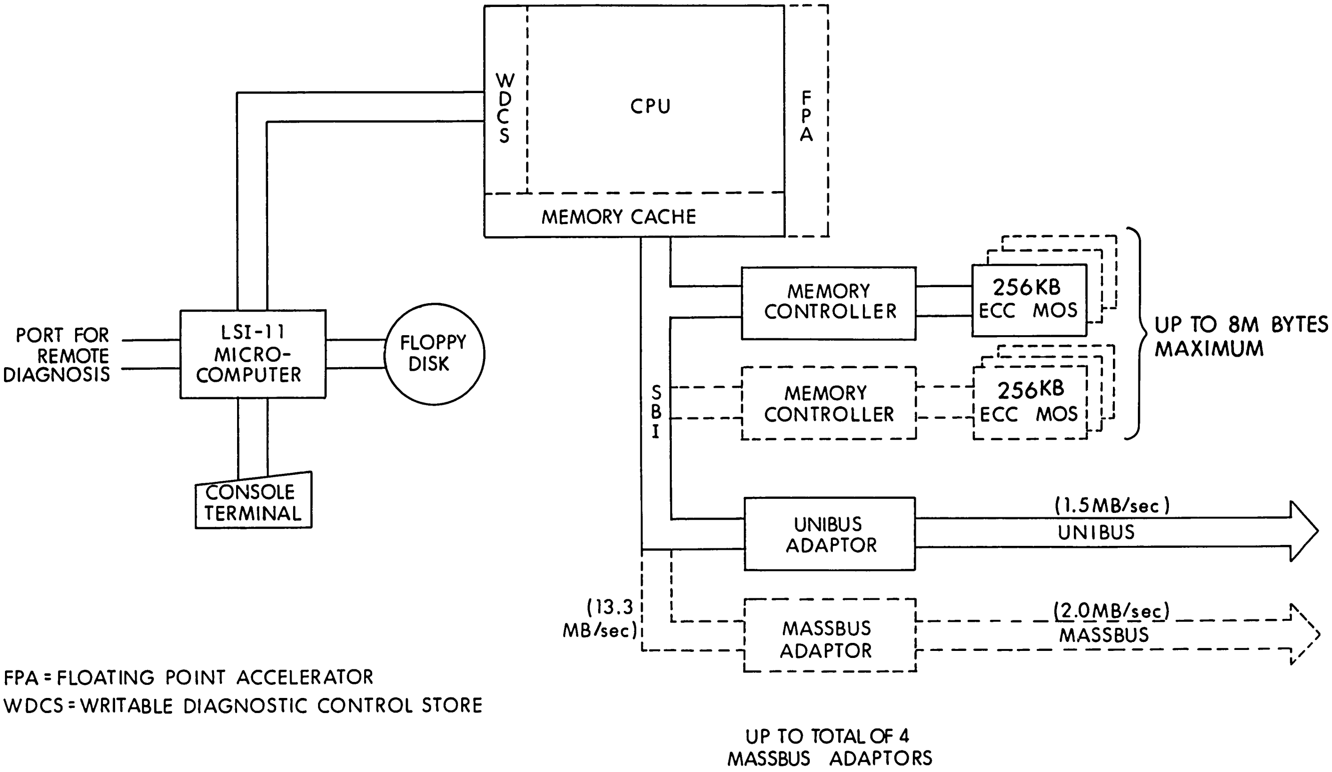


Figure 4-1
Processor Components

The next few pages introduce some of the concepts that concern assembly language programmers in general, including addressing, data types, instruction sets, and other programming aspects of the processor. Further details on these programming characteristics follow this introduction.

Process Virtual Address Space

Most data are located in memory using the address of an 8-bit byte. The programmer uses a 32-bit virtual address to identify a byte location. This address is called a *virtual* address because it is not the real address for a physical memory location. It is translated into a real address by the processor under operating system control. A virtual address is not a unique address of a location in memory, as are physical memory addresses. Two programs using the same virtual address might refer to two different physical memory locations, or refer to the same physical memory location using different virtual addresses.

The set of all possible 32-bit virtual addresses is called virtual address space. It can be viewed as an array of byte "locations" labelled from zero to 2^{32} minus one, an array that is approximately four billion bytes in length. This address space is divided into sets of virtual addresses designated for certain uses. The set of virtual addresses designated for use by a process, including an image it executes, is one half of the total virtual address space. Addresses in the remaining half of virtual address space are used to refer to locations maintained and protected by the operating system.

Instruction Sets

At any one time, the processor's instruction interpretation hardware can be set to either of two modes: native mode or compatibility mode. In native mode the processor executes a large set of variable-length instructions, recognizes a variety of data types, and uses sixteen 32-bit general purpose registers. In compatibility mode the processor executes a set of PDP-11 instructions, recognizes integer data, and uses eight 16-bit general purpose registers. While native mode is the primary instruction execution state of the machine and compatibility mode the secondary state, their instruction sets are closely related, and their programming characteristics are very similar. A user process can execute both native mode images and compatibility mode images.

A native instruction consists of an operation code (opcode) and zero or more operands, which are described by data type and addressing mode. The native instruction set is based on over 200 different kinds of operations, most of which can be applied to any one of nine types of data, which can be addressed in any one of several ways. Thus, the native instruction set offers a tremendous number of instructions from which to choose.

In spite of the large number of instructions, the native instruction set is a natural programming language that is very easy to learn. Many of the instructions correspond directly to high-level language statements, and the assembler mnemonics are readily associated with the instruction function.

To choose the appropriate instruction, it is only necessary to become familiar with the operations, data types, and addressing modes. For example, the ADD operation can be applied to any of several sizes of integer, floating point, or packed decimal operands, and each operand can be addressed directly in a register, directly in memory, or indirectly through pointers stored in registers or memory locations.

Registers and Addressing Modes

A register is a location within the processor that can be used for temporary data storage and addressing. The assembly language programmer has sixteen 32-bit general registers available for use with the native instruction set. Some of these registers have special significance. For example, one register is designated as the Program Counter, and it contains the address of the next instruction to be executed. Three general registers are designated for use with routine linkages: the Stack Pointer, the Argument Pointer, and the Frame Pointer.

An instruction operand can be located in main memory, in a general register, or in the instruction stream itself. The way in which you choose to specify an operand location is called the operand *addressing mode*. The processor offers a variety of addressing modes and addressing mode optimizations. There is one addressing mode that locates an operand in a register. There are six addressing modes that locate an operand in memory using a register to:

- point to the operand
- point to a table of operands
- point to a table of operand addresses

There also are six addressing modes that are indexed modifications of the addressing modes that locate an operand in memory. Finally, there are two addressing modes that identify the location of the operand in the instruction stream: one for constant data, and one for branch instruction addresses.

Data Types

The data type of an instruction operand identifies how many bits of storage are to be treated as a unit, and what the interpretation of that unit is. The processor's native instruction set recognizes four primary data types: integer, floating point, packed decimal, and character string. For each of these data types, your selection of operation immediately tells the processor the size of the data and its interpretation. In addition, the processor can also manipulate a fifth data type, the bit field, in which you define the size of the field and its relative position.

There are several variations on the four primary data types. Table 4-1 provides a summary of the data types available, some of which are illustrated in Figure 4-2. Integer data are stored as a binary value. You have the choice of storing an integer as a byte, word, longword, or, in some cases, as a quadword. A byte is eight bits, a word is two bytes, a longword is four bytes, and a quadword is eight bytes. The processor can interpret an integer as either a signed (two's complement) value or an unsigned value. The sign is determined by the high-order bit.

Floating point values are stored using a signed 8-bit excess-128 exponent and a binary normalized fraction, using 4-byte and 8-byte formats identical to those of the PDP-11. The normalization bit is not represented, to provide an effective 24-bit fraction in the 4-byte format, and an effec-

Table 4-1
Data Types

DATA TYPE	SIZE	RANGE (decimal)	
Integer		Signed	Unsigned
Byte	8 bits	-128 to +127	0 to 255
Word	16 bits	-32768 to +32767	0 to 65535
Longword	32 bits	-2^{31} to $+2^{31}-1$	0 to $2^{32}-1$
Quadword	64 bits	-2^{63} to $+2^{63}-1$	0 to $2^{64}-1$
Floating Point		$\pm 2.9 \times 10^{-37}$ to 1.7×10^{38}	
Floating	32 bits	approximately seven decimal digits precision	
Double Floating	64 bits	approximately sixteen decimal digits precision	
Packed Decimal String	0 to 16 bytes (31 digits)	numeric, two digits per byte sign in low half of last byte	
Character String	0 to 65535 bytes	one character per byte	
Variable-length Bit Field	0 to 32 bits	dependent on intrepretation	

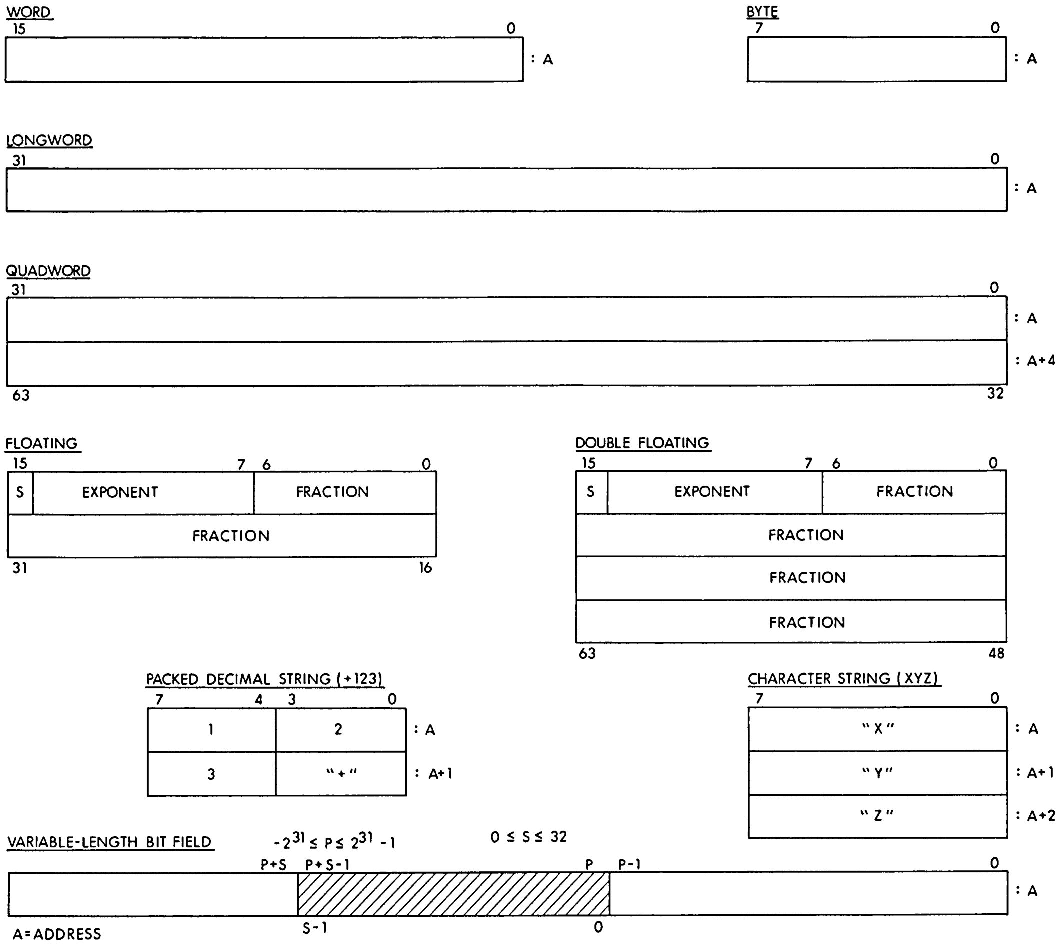


Figure 4-2
Data Type Representations

tive 56-bit fraction in the 8-byte format. The 4-byte format, called simply floating, provides approximately seven decimal digits of precision, while the 8-byte format, called double floating, provides approximately 16 decimal digits of precision.

Packed decimal data are stored in a string of bytes. Each byte is divided into two 4-bit nibbles. One decimal digit is stored in each nibble. The first, or high-order, digit is stored in the high-order nibble of the first byte, the second digit is stored in the low-order nibble of the first byte, the third digit is stored in the high-order nibble of the second byte, and so on. The sign of the number is stored in the low-order nibble of the last byte of the string.

Character data are simply a string of bytes containing any binary data, for example, ASCII codes. The first character in the string is stored in the first byte, the second character is stored in the second byte, and so on. A character string that contains ASCII codes for decimal digits is called a numeric string.

The address of any data item is the address of the first byte in which the item resides. All integer, floating point, packed decimal, and character data can be stored starting on an arbitrary byte boundary. A bit field, however, does not necessarily start on a byte boundary. A field is simply a set of contiguous bits (0-32) whose starting bit location is identified relative to a given byte address. The native instruction set can interpret a bit field as a signed or unsigned integer.

Stacks, Subroutines, and Procedures

A stack is an array of consecutively addressed data items that are referenced on a last-in, first-out basis using a general register. Data items are added to and removed from the low address end of the stack. A stack grows toward lower addresses as you add items, and shrinks toward higher addresses as you remove items.

You can create a stack anywhere in your program's address space and use any register to point to the current item on the stack. The operating system, however, automatically reserves portions of each process address space for stack data structures. User software references its stack data structure, called the **user stack**, through a general register designated as the Stack Pointer. When the user runs a program image, the operating system automatically provides the address of the area designated for the user stack.

A stack is an extremely powerful data structure because it can be used to pass arguments to routines efficiently. In particular, the stack structure enables you to write reentrant routines because the processor can handle routine linkages automatically using the Stack Pointer. Routines can also be recursive because arguments can be saved on the stack for each successive call of the same routine.

The processor provides two kinds of routine call instructions: those for **subroutines**, and those for **procedures**. In general, a subroutine is a routine entered using a Jump to Subroutine or Branch to Subroutine instruction, while a procedure is a routine entered using a Call instruction.

The processor provides more elaborate routine linkages for procedures than for subroutines. The processor automatically saves and restores the contents of registers you want preserved across procedure calls. The processor provides two methods for passing argument lists to called procedures: by passing the arguments on the stack, or by passing the address of the arguments elsewhere in memory. The processor also constructs a "journal" of procedure call nesting by using a general register as a pointer to the place on the stack where a procedure has its linkage data. This record of each procedure's stack data, known as its **stack frame**, enables proper returns from procedures even when a procedure leaves data on the stack. In addition, user and operating system software can use the stack frame to trace back through nested calls to handle errors or debug programs.

Condition Codes

A user program can test the outcome of an arithmetic or logical operation. The processor provides a set of condition codes and branch instructions for this purpose. The condition codes indicate whether the previous arithmetic or logical operation produced a negative or zero result, or whether there was a carry or borrow, or an overflow. There are a variety of branch on condition instructions: those for overflow and carry or borrow, and those for signed and unsigned relational tests.

Exceptions

There are some situations in which you may not want to have to test the outcome of an operation. The processor recognizes many events for which you do not have to test, and automatically changes the normal flow of your program when they occur. These events, called **exceptions**, are the direct result of executing a specific instruction. Exceptions also include errors automatically detected by the processor, such as improperly formed instructions.

All exceptions trap to operating system software. There are essentially no fatal exceptions. All exceptions either wait for the instruction that caused them to complete before trapping or they restore the processor to the state it was in just prior to executing the instruction that caused the exception. In either case, the instruction can be retried after the cause of the exception is cleared. Depending on the exception, it may be desirable to correct the situation and continue. If not, the operating system issues an appropriate message and aborts the instruction stream in progress. To continue, you can request the operating system software to start execution of a condition handler automatically when an exception occurs.

USER PROGRAMMING ENVIRONMENT

A process context includes the definition of the virtual address space in which it executes an image. An image executing within a process context controls its execution through the use of one of the instruction sets, the general purpose registers, and the Processor Status Word. These hardware resources are discussed in detail in the following sections.

Process Virtual Address Space Structure

As mentioned earlier, certain sets of virtual addresses in virtual address space are designated for particular uses. To begin with, the processor and operating system provide a multiprogramming environment by dividing virtual address space into two halves: one half for addressing context-dependent code and data, and one half for addressing context-independent code and data.

The first half is called **per-process space** (or more simply, "process space"), which is capable of addressing approximately two billion bytes. An image executing in the context of a process and the operating system on behalf of the process use addresses in process space to refer to code and data particular to that process context. A process can not represent virtual addresses in any process space but its own. Thus, code and data belonging to a process are automatically protected from other processes in the system.

The second half of virtual address space is called **system space**. The operating system assigns the meanings to addresses in system space. The significance of any address in system space is the same for every process, independent of process context. Although most locations referred to by system space addresses are protected from access by user images, if two images executing in different process contexts do use an address in system space, the address always refers to the same location in memory.

Process space is further subdivided into two regions. Addresses in the first region, called the **program region**, are used to identify the location of image code and data. Addresses in the second region, called the **control region**, are used to refer to stacks and other temporary program image and permanent process control information maintained by the operating system on behalf of the process. Program region addresses are allocated from address 0 up, and control region addresses are allocated from address 2^{31}-1 down.

System space is also subdivided into two regions. The operating system assigns the **system region** addresses for linkages to its service procedures, for memory management data, and for I/O processing routines. The second region is presently unused.

General Registers

Instruction operands are often either stored in the processor's general registers or accessed through them. The sixteen 32-bit programmable general registers are labelled R0 through R15 (in decimal). Registers can be used for temporary storage, accumulators, base registers, and index registers. A base register contains the address of the base of a software data structure such as a table or queue, and an index register contains a logical offset into a data structure.

Whenever a register is used to contain data, the data are stored in the register in the same format as would appear in memory. If a quadword or double floating datum is stored in a register, it is actually stored in two adjacent registers. For example, storing a double floating number in register R7 loads both R7 and R8.

Some registers have special significance depending on the instruction being executed. Registers R12 through R15 have special significance for many instructions, and therefore have special labels. These special registers are:

- the Program Counter (PC or R15), which contains the address of the next byte to be processed in the instruction stream.
- the Stack Pointer (SP or R14), which contains the address of the base (or top) of a stack maintained for subroutine and procedure calls.
- the Frame Pointer (FP or R13), which contains the address of the base of a software data structure stored on the stack called the stack frame, which is maintained for procedure calls.
- the Argument Pointer (AP or R12), which contains the address of the base of a software data structure called the argument list, which is maintained for procedure calls.

In addition, the first six registers, R0 through R5, have special significance for character and packed decimal string instructions, and the Polynomial Evaluation instruction. These instructions use these registers to store temporary results and, upon completion, leave results in the registers that a program can use as the operands of subsequent instructions.

A register's special significance does not preclude its use for other purposes, except for the Program Counter. The Program Counter can not be used as an accumulator, as a temporary register, or as an index register. In general, however, most users do not use the Stack Pointer, Argument Pointer, or Frame Pointer for purposes other than those designated.

Addressing Modes

The processor's addressing modes allow almost any operand to be stored in a register or in memory, or as an immediate constant. Figure 4-3 summarizes the addressing modes. There are seven basic addressing modes that use the general registers to identify the operand location. They include:

- Register mode, in which the register contains the operand.
- Register Deferred mode, in which the register contains the address of the operand.
- Autodecrement mode, in which the contents of the register are first decremented by the size of the operand, and then used as the address of the operand. The size of the operand (in bytes) is given by the data type of the instruction operand, and depends on the instruction. For example, the Clear Word instruction gives a size of two, since there are two bytes per word.
- Autoincrement mode, in which the contents of the register are used as the address of the operand, and then incremented by the size of the operand. If the Program Counter is the specified register, the mode is called Immediate mode.
- Autoincrement Deferred mode, in which the contents of the register are used as the address of a location in memory containing the address of the operand, and then are incremented by four (the size of an address). If the Program Counter is the specified register, the mode is called Absolute mode.
- Displacement mode, in which the value stored in the register is used as a base address. A byte, word, or longword signed constant is added to the base address, and the resulting sum is the effective address of the operand.

Literal (Immediate)	{S↑ I↑} # *constant*	
Register	R*n*	
Register Deferred Autodecrement	(R*n*) -(R*n*)	Indexed [Rx]
Autoincrement	(R*n*) +	
Autoincrement Deferred (Absolute)	@ (R*n*) + @ # *address*	
Displacement	{B↑ W↑ L↑} *displacement* (R*n*) *address*	
Displacement Deferred	@{B↑ W↑ L↑} *displacement* (R*n*) *address*	

n = 0 through 15
x = 0 through 14

Figure 4-3
Addressing Modes: Assembler Syntax

- Displacement Deferred mode, in which the value stored in the register is used as the base address of a table of addresses. A byte, word, or longword signed constant is added to the base address, and the resulting sum is the address of the location that contains the actual address of the operand.

Of these seven basic modes, all except register mode can be modified by an index register. When an index register is used with a basic mode to identify an operand, the addressing mode is the name of the basic mode with the suffix "indexed." For example, the indexed addressing mode for register deferred is called "register deferred indexed" mode. In addition to the seven basic addressing modes that use registers, therefore, the processor recognizes six indexed addressing modes.

In an indexed addressing mode, one register is used to compute the base address of a data structure, and the other register is used to compute an index offset into the data structure. To obtain the operand's effective address in an indexed addressing mode, the processor: 1) computes the base operand address provided by one of the basic addressing modes (except register mode), 2) takes the value stored in the index register and multiplies it by the given operand size, and 3) adds the resultant value to the operand address. The index register contents are not affected and can be used for subsequent processing operations.

The processor also provides literal mode addressing, in which an unsigned 6-bit field in the instruction is interpreted as an integer or floating point constant.

The variety of addressing modes enables the assembly language programmer to write, and high-level language compilers to produce, very compact code. For example, literal mode is a very efficient way to specify small constants. The 6-bit field is interpreted as an integer when used with integer operations, and can therefore express the constants 0 through 63 decimal. The 6-bit field is interpreted as a floating point constant when used in floating point operations, where three bits express an exponent and three a fraction.

The autoincrement and autodecrement modes enable you to automatically step through tables. Displacement mode lets you generate offsets into a table, with a choice of either short or long displacements. The deferred modes enable you to maintain tables of operand addresses instead of the operands themselves.

The indexed addressing modes allow you to index into tables with a step size automatically determined by the operation. As in autoincrement and autodecrement addressing, the index is calculated in the context of the operand data type. This means that you can easily access several tables of differing data types using the same index key.

Furthermore, because the indexed addressing modes enable you to specify the base operand address using any mode that generates an actual address (that is, any mode except register or literal), you have the ability to construct double indexing. You can select the base address from a table of base addresses using displacement deferred mode, and then use an index register to provide the offset into the particular table selected.

Thus the processor's addressing modes allow considerable flexibility in the arrangement and processing of data structures. A data structure's design does not have to be tied to its processing method for efficiency.

Program Counter

A native mode instruction has a variable-length format, and instructions are thought of as byte aligned. A variable-length format not only makes code more compact, it means that the instruction set can be extended easily. The opcode for the operation is generally a single byte, and it is followed by zero to six operand specifiers, depending on the instruction. An operand specifier can be one or several bytes long, depending on the addressing mode. Figure 4-4 illustrates the representation of an instruction as a string of bytes. Just before the processor begins to execute an instruction, the Program Counter contains the address of the first byte of the next instruction. The way in which the Program Counter is updated is totally transparent to the programmer.

The Program Counter itself can be used to identify operands. The assembler translates many types of operand references into addressing modes using the Program Counter. Autoincrement mode using the Program Counter, or **immediate mode,** is used to specify in-line constants other than those available with literal mode addressing. Autoincrement deferred mode using the Program Counter, or **absolute mode,** is used to reference an absolute address. Displacement and displacement deferred modes using the Program Counter are used to specify an operand using an offset from the current location.

Addressing using the Program Counter enables you to write position independent code. Position independent code can be executed anywhere in virtual address space after it has been linked, since program linkages can be identified as absolute locations in virtual address space and all other addresses can be identified relative to the current instruction.

The Stack Pointer, Argument Pointer and Frame Pointer

The Stack Pointer is a register specifically designated for use with stack structures. To place items on a stack, you can use autodecrement mode addressing through the Stack Pointer, and to remove them, you can use autoincrement mode. You can reference and modify the top element on a stack without removing it by using register deferred mode, and can reference other elements of the stack using displacement mode addressing.

The processor designates Register 14 as the Stack Pointer for use with both the subroutine Branch or Jump instructions, and the procedure Call instructions. On routine entry, the processor automatically saves the address of the instruction that follows the routine call on the stack. It uses the Program Counter and the Stack Pointer to perform the operation. Before entering the subroutine, the Program Counter contains the address of the instruction following the Branch, Jump, or Call instruction. The Stack Pointer contains the address of the last item on the stack. The processor pushes the contents of the Program Counter on the stack. On return from a subroutine, the processor automatically restores the Program Counter by popping the return address off the stack.

Also for the procedure Call instructions, the processor designates Register 12 as an Argument Pointer, and Register 13 as a Frame Pointer. The Argument Pointer is used to pass the address of the argument list to a called procedure, and the Frame Pointer is used to keep track of nested Call instructions.

An argument list is a formal data structure that contains the arguments required by the procedure you are calling. Arguments may be actual values, addresses of data structures, or addresses of other procedures. You can pass an argument list in either of two ways: by passing only its

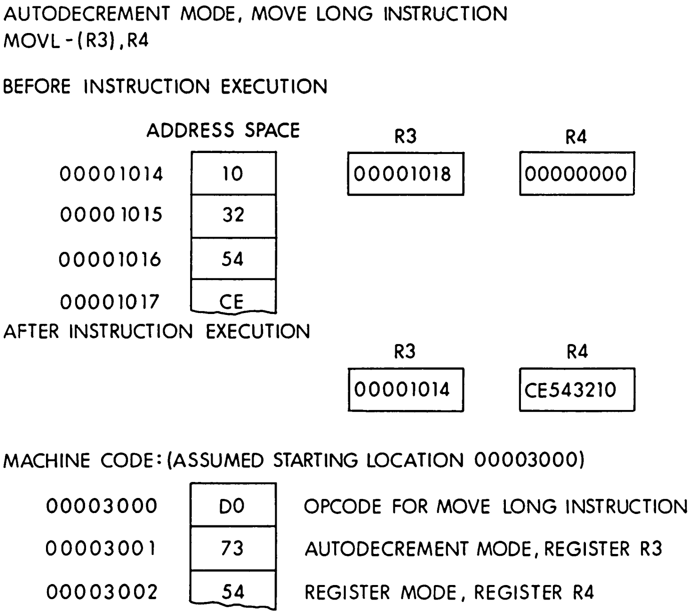

AUTODECREMENT MODE, MOVE LONG INSTRUCTION
MOVL -(R3),R4
BEFORE INSTRUCTION EXECUTION
ADDRESS SPACE
00001014
10
00001015
32
00001016
54
00001017
CE
R3
00001018
R4
00000000
AFTER INSTRUCTION EXECUTION
R3
00001014
R4
CE543210
MACHINE CODE: (ASSUMED STARTING LOCATION 00003000)
00003000
D0
OPCODE FOR MOVE LONG INSTRUCTION
00003001
73
AUTODECREMENT MODE, REGISTER R3
00003002
54
REGISTER MODE, REGISTER R4

Figure 4-4
Instruction Representation

address, or by passing the entire list on the user stack. The first method is used to pass long argument lists, or lists that you want to preserve. The second method is generally used when calling procedures that do not require arguments, or when you are building an argument list dynamically.

When you issue a procedure Call instruction, the processor uses the Argument Pointer to pass arguments to the procedure. If you passed arguments on the stack, the processor automatically pops the arguments off for you on return from the procedure.

The importance of the way the Call instructions work is that nested calls can be traced back to any previous level. The Call instructions always keep track of nested calls by using the Frame Pointer register. The Frame Pointer contains the address on the stack of the items pushed on the stack during the procedure call. The set of items pushed on the stack during a procedure call is known as a **call frame** or **stack frame**. Since the previous contents of the Current Frame register are saved in each call frame, the nested frames form a linked data structure which can be unwound to any level when an error or exception condition occurs in any procedure.

Processor Status Word

The Processor Status Word (a portion of the Processor Status Longword) is a special processor register that a program uses to check its status and to control synchronous error conditions. The Processor Status Word, illustrated in Figure 4-5, contains two sets of bit fields:

- the condition codes
- the trap enable flags

The condition codes indicate the outcome of a particular logical or arithmetic operation. For example, the Subtract instruction sets the Negative bit if the result of the subtraction operation produced a negative number, and it sets the Zero bit if the result produced zero. You can then use the Branch on Condition instructions to transfer control to a code sequence that handles the condition.

There are two kinds of traps that concern the user process: trace traps and arithmetic traps. The trace trap is used by debugging programs or performance evaluators. Arithmetic traps include:

- integer, floating point, or decimal string overflow, in which the result was too large to be stored in the given format
- integer, floating point, or decimal string divide by zero, in which the divisor supplied was zero
- floating point underflow, in which the result was too small to be expressed in the given format

Of these arithmetic traps, you have two choices in the method you can use to handle integer overflow, floating underflow, and decimal string overflow. If you clear the trap enable bits in the Processor Status Word, the processor will ignore integer and decimal string overflow and floating underflow. If you want to check for these conditions, you can either test the condition codes yourself (using the Branch On Condition instructions), or you can enable the trap bits. If you enable the trap bits, the processor treats integer and decimal string overflow and floating underflow as exceptions. In any case, floating overflow and divide by zero traps are always enabled.

Handling Exceptions

When an exception occurs, the processor immediately saves the current state of execution and traps to the operating system. The operating system automatically searches for a procedure that wants to handle the exception. Procedures that respond to exceptions are called **condition handlers**. You can declare a condition handler for an entire image and for each individual procedure called. In addition, because the processor keeps track of nested calls using the Frame Pointer register, it is possible to declare condition handlers for procedures that call other procedures in which exceptions might occur. The operating system automatically traces back through call frames to find a condition handler that wants to handle an exception that occurs.

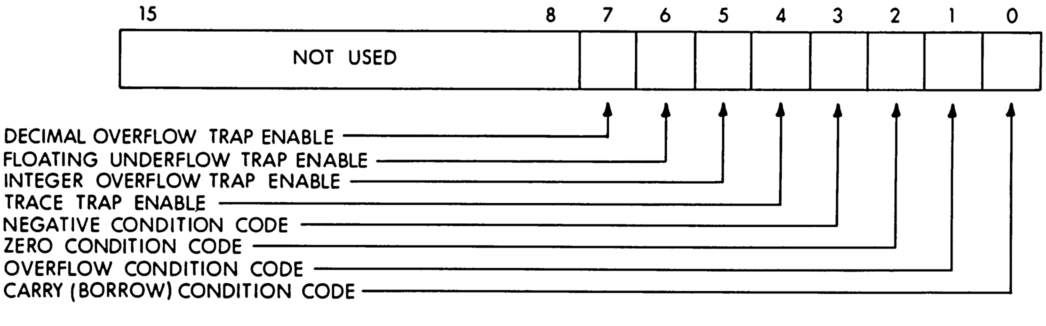

15
8
7
6
5
4
3
2
1
0
NOT USED
DECIMAL OVERFLOW TRAP ENABLE
FLOATING UNDERFLOW TRAP ENABLE
INTEGER OVERFLOW TRAP ENABLE
TRACE TRAP ENABLE
NEGATIVE CONDITION CODE
ZERO CONDITION CODE
OVERFLOW CONDITION CODE
CARRY (BORROW) CONDITION CODE

Figure 4-5
Processor Status Word

NATIVE INSTRUCTION SET

The instruction set that the processor executes is selected under operating system control to either native mode or compatibility mode. The native mode instruction set is based on over 200 different opcodes. The opcodes can be grouped into classes based on their function and use. Instructions used to manipulate the general data types include:

- integer and floating point instructions
- packed decimal instructions
- character string instructions
- bit field instructions

Instructions that are used to manipulate special kinds of data include:

- queue manipulation instructions
- address manipulation instructions
- user-programmed general register control instructions

Instructions that provide basic program flow control, and enable you to call procedures are:

- branch, jump and case instructions
- subroutine call instructions
- procedure call instructions

Table 4-2 lists the basic instruction operations in order by these classifications. Instructions that enable operating system procedures to provide user mode processes with services requiring privilege are listed in the table, but discussed in the system programming environment section. Instructions that are singular in the functions they provide are listed last. The following paragraphs describe the functions of most of the instructions within each class. For further information on the instruction set, refer to the appropriate VAX-11/780 Handbook.

Table 4-2
Summary Instruction Set

Integer and Floating Point Logical Instructions	
MOV_	Move (B,W,L,F,D,Q)*
MNEG_	Move Negated (B,W,L,F,D)
MCOM_	Move Complemented (B,W,L)
MOVZ__	Move Zero-Extended (BW,BL,WL)
CLR_	Clear (B,W,L=F,Q=D)
CVT__	Convert (B,W,L,F,D)(B,W,L,F,D)
CVTR_L	Convert Rounded (F,D) to Longword
CMP_	Compare (B,W,L,F,D)
TST_	Test (B,W,L,F,D)
BIS_2	Bit Set (B,W,L) 2-Operand
BIS_3	Bit Set (B,W,L) 3-Operand
BIC_2	Bit Clear (B,W,L) 2-Operand
BIC_3	Bit Clear (B,W,L) 3-Operand
BIT_	Bit Test (B,W,L)
XOR_2	Exclusive OR (B,W,L) 2-Operand
XOR_3	Exclusive OR (B,W,L) 3-Operand
ROTL	Rotate Longword

Integer and Floating Point Arithmetic Instructions	
INC_	Increment (B,W,L)
DEC_	Decrement (B,W,L)
ASH_	Arithmetic Shift (L,Q)
ADD_2	Add (B,W,L,F,D) 2-Operand
ADD_3	Add (B,W,L,F,D) 3-Operand
ADWC	Add with Carry
ADAWI	Add Aligned Word Interlocked
SUB_2	Subtract (B,W,L,F,D) 2-Operand
SUB_3	Subtract (B,W,L,F,D) 3-Operand
SBWC	Subtract with Carry
MUL_2	Multiply (B,W,L,F,D) 2-Operand
MUL_3	Multiply (B,W,L,F,D) 3-Operand
EMUL	Extended Multiply
DIV_2	Divide (B,W,L,F,D) 2-Operand
DIV_3	Divide (B,W,L,F,D) 3-Operand
EDIV	Extended Divide
EMOD_	Extended Modulus (F,D)
POLY_	Polynomial Evaluation (F,D)

*B = byte, W = word, L = longword, F = floating, D = double floating, Q = quadword.

Packed Decimal Instructions	
MOVP	Move Packed
CMPP3	Compare Packed 3-Operand
CMPP4	Compare Packed 4-Operand
ASHP	Arithmetic Shift Packed and Round
ADDP4	Add Packed 4-Operand
ADDP6	Add Packed 6-Operand
SUBP4	Subtract Packed 4-Operand
SUBP6	Subtract Packed 6-Operand
MULP	Multiply Packed
DIVP	Divide Packed
CVTLP	Convert Long to Packed
CVTPL	Convert Packed to Long
CVTPT	Convert Packed to Trailing
CVTTP	Convert Trailing to Packed
CVTPS	Convert Packed to Separate
CVTSP	Convert Separate to Packed
EDITPC	Edit Packed to Character String

Character String Instructions	
MOVC3	Move Character 3-Operand
MOVC5	Move Character 5-Operand
MOVTC	Move Translated Characters
MOVTUC	Move Translated Until Character
CMPC3	Compare Characters 3-Operand
CMPC5	Compare Characters 5-Operand
LOCC	Locate Character
SKPC	Skip Character
SCANC	Scan Characters
SPANC	Span Characters
MATCHC	Match Characters

Variable-Length Bit Field Instructions	
EXTV	Extract Field
EXTZV	Extract Zero-Extended Field
INSV	Insert Field
CMPV	Compare Field
CMPZV	Compare Zero-Extended Field
FFS	Find First Set
FFC	Find First Clear

Table 4-2 (cont)
Summary Instruction Set

Index Instruction	
INDEX	Compute Index

Queue Instructions	
INSQUE	Insert Entry in Queue
REMQUE	Remove Entry from Queue

Address Manipulation Instructions	
MOVA_	Move Address (B,W,L=F,Q=D)
PUSHA_	Push Address (B,W,L=F,Q=D) on Stack

General Register Manipulation Instructions	
PUSHL	Push Longword on Stack
PUSHR	Push Registers on Stack
POPR	Pop Registers from Stack
MOVPSL	Move from Processor Status Longword
BISPSW	Bit Set Processor Status Word
BICPSW	Bit Clear Processor Status Word

Unconditional Branch and Jump Instructions	
BR_	Branch with (Byte, Word) Displacement
JMP	Jump

Branch on Condition Code	
BLSS	Less Than
BLSSU	Less than Unsigned
BLEQ	Less than or Equal
BLEQU	Less than or Equal Unsigned
BEQL	Equal
(BEQLU)	(Equal Unsigned)
BNEQ	Not Equal
(BNEQU)	(Not Equal Unsigned)
BGTR	Greater than
BGTRU	Greater than Unsigned
BGEQ	Greater than or Equal
BGEQU	Greater than or Equal Unsigned
(BCC)	(Carry Cleared)
BVS	Overflow Set
BVC	Overflow Clear

Branch on Bit	
BLB_	Branch on Low Bit (Set, Clear)
BB_	Branch on Bit (Set, Clear)
BBS_	Branch on Bit Set and (Set, Clear) bit
BBC_	Branch on Bit Clear and (Set, Clear) bit
BBSSI	Branch on Bit Set and Set bit Interlocked
BBCCI	Branch on Bit Clear and Clear bit Interlocked

Loop and Case Branch	
ACB_	Add, Compare and Branch (B,W,L,F,D)
AOBLEQ	Add One and Branch Less Than or Equal
AOBLSS	Add One and Branch Less Than
SOBGEQ	Subtract One and Branch Greater Than or Equal
SOBGTR	Subtract One and Branch Greater Than
CASE_	Case on (B,W,L)

Subroutine Call and Return Instructions	
BSB_	Branch to Subroutine with (B,W) Displacement
JSB	Jump to Subroutine
RSB	Return from Subroutine

Procedure Call and Return Instructions	
CALLG	Call Procedure with General Argument List
CALLS	Call Procedure with Stack Argument List
RET	Return from Procedure

Protected Procedure Call and Return Instructions	
CHM_	Change Mode to (Kernel, Executive, Supervisor, User)
REI	Return from Exception or Interrupt
PROBER	Probe Read
PROBEW	Probe Write

Privileged Processor Register Control Instructions	
SVPCTX	Save Process Context
LDPCTX	Load Process Context
MTPR	Move to Process Register
MFPR	Move from Processor Register

Special Function Instructions	
CRC	Cyclic Redundancy Check
BPT	Breakpoint Fault
XFC	Extended Function Call
NOP	No Operation
HALT	Halt

Integer and Floating Point Instructions

The logical and arithmetic processor instructions illustrate how the opcodes, data types, and addressing modes can be combined in an instruction. Most of the operations provided for integer data are also provided for floating point and packed decimal data. Exceptions are the strictly logical operations for integer data (such as bit clear, bit set, complement), the multiword arithmetic instructions for integer data (such as Add/Subtract with Carry and Extended Multiply and Extended Divide), and the Extended Modulus and Polynomial instructions for floating point data.

The arithmetic instructions include both 2-operand and 3-operand forms that eliminate the need to move data to and from temporary operands. The 2-operand instructions store the result in one of the two operands, as in "Set A equal to A plus B." The 3-operand instructions effectively implement the high-level language statements in which two different variables are used to calculate a third, such as "Set C equal to A plus B." The 3-operand instructions are applicable to both integer and floating point data, and equivalent instructions exist for packed decimal data.

To illustrate the instruction set and addressing modes, consider the FORTRAN language statement:

A(I) = B(I) * C(I)

where A, B, and C are statically allocated REAL*4 arrays and I is INTEGER*4. A code sequence that performs this operation is:

```
MOVL     I,R0                 ;Move the longword I
                              ;to a register
MULF3    B[R0],C[R0],A[R0]    ;3-operand floating
                              ;multiply
```

The same code applies if A, B, and C are REAL*8, INTEGER*4, INTEGER*2, or even INTEGER*1 data types: the MULF3 instruction is simply changed to MULD3, MULL3, MULW3, or MULB3, respectively.

If arrays A, B, and C are dynamically allocated arrays, the code sequence could be:

```
MOVL     I,R0
MULF3    Bdisp(FP)[R0],Cdisp(FP)[R0],Adisp(FP)[R0]
```

If A, B, and C are arguments to a procedure, the code could be:

```
MOVL     I,R0
MULF3    @Bargptr(AP)[R0],@Cargptr(AP)[R0],
          @Aargptr[R0]
```

In fact, the locations of A, B, and C can be arbitrarily selected. For example, combining the above, if A is statically allocated, B dynamically allocated, and C an argument, then the code sequence could be:

```
MOVL     I,R0
MULF3    Bdisp(FP)[R0],@Cargptr(AP)[R0],A[R0]
```

Some of the arithmetic instructions are used for extending the accuracy of repeated computations. The Extended Multiply (EMUL) instruction takes longword integer arguments and produces a quadword result. The instruction effectively implements a high-level language statement such as "Set D equal to (A times B) plus C." The Extended Divide (EDIV) instruction divides a quadword integer by a longword and produces a longword quotient and a longword remainder.

The Extended Modulus (EMOD) instructions multiply a floating point number with an extended precision floating point number (extended by eight bits for an effective 9 or 19 digits of accuracy) and returns the integer portion and the fractional portion separately. This instruction is particularly useful for preserving the precision of input throughout trigonometric and exponential function evaluation.

The Polynomial Evaluation (POLY) instructions evaluate a polynomial from a table of coefficients using Horner's method. This instruction is used extensively in the high-level languages' math library for operations such as sine and cosine.

Packed Decimal Instructions

Many of the operations for integer and floating point data also apply to packed decimal strings. They include:

- Move Packed (MOVP) for copying a packed decimal string from one location to another, and Arithmetic Shift Packed (ASHP) for scaling a packed decimal up or down by a given power of 10 while moving it, and optionally rounding the value.
- Compare Packed (CMPP) for comparing two packed decimal strings. Compare Packed has two variations: a 3-operand (CMPP3) instruction for strings of equal length, and a 4-operand instruction (CMPP4) for strings of differing lengths.
- Convert Instructions, including Convert Long to Packed (CVTLP), Convert Packed to Long (CVTPL), Convert Packed to numeric with Trailing sign (CVTPT), Convert numeric with Trailing sign to Packed (CVTTP), Convert Packed to numeric with Separate overpunched sign (CVTPS), and Convert numeric with Separate overpunched sign to Packed (CVTSP). These instructions enable you to convert our packed decimal format to commonly used numeric formats. Numeric with trailing sign allows various sign encodings including zoned and overpunched.
- Add Packed (ADDP) and Subtract Packed (SUBP) for adding or subtracting two packed decimal strings, with the option of replacing the addend or subtrahend with the result (ADDP4 and SUBP4), or storing the result in a third string (ADDP6 or SUBP6).
- Multiply Packed (MULP) and Divide Packed (DIVP) for multiplying or dividing two packed decimal strings and storing the result in a third string.

In addition, the packed decimal instructions include a special packed decimal string to character string conversion instruction that provides output formatting: the Edit instruction.

Edit Instruction

The Edit Packed to Character String (EDITPC) instruction supplies formatted numeric output functions. The instruction converts a given packed decimal string to a character string using selected pattern operators. The pattern operators enable you to create numeric output fields with any of the following characteristics:

- leading zero fill
- leading zero protection
- leading asterisk fill protection
- a floating sign
- a floating currency symbol
- special sign representations
- insertion characters
- blank when zero

Character String Instructions

The character string instructions operate on strings of bytes. They include:

- move string instructions, with translation options
- string compare instructions
- single character search instructions
- substring search instructions

There are two basic forms of Move instructions for character strings. The Move Character instructions (MOVC3 and MOVC5) simply copy character strings from one location to another. They are optimized for block transfer operations. The 5-operand variation enables you to supply a fill character that the instruction uses to pad out the destination location to a given size.

The Move Translated Characters (MOVTC) and Move Translated Until Character (MOVTUC) instructions actually create new character strings. You supply a string which the instruction uses as a list of offsets into a translation table. The instruction selects characters from the table in the order that the offset list points to the table. The MOVTC instruction allows you to supply a fill character that the instruction uses to pad out the resultant string to a given size with an arbitrary character. The MOVTUC instruction allows you to supply any number of escape characters. When the next offset points to an escape character in the table, translation stops.

The Compare Characters (CMPC) instructions provide character-by-character byte string compares. CMPC has a 3-operand form and a 5-operand form. Both instructions compare two strings from beginning to end and let you know when they reach the first character that is different between the strings, or when they get to the end of either string. The 5-operand variation lets you supply a fill character which it uses to effectively pad out a string when comparing it with a longer one.

The Locate Character (LOCC) and Skip Character (SKPC) instructions are search instructions for single characters within a string. LOCC searches a given string for a character that matches the search character you supply. This is useful, for example, when searching for the delimiter at the end of a variable-length string. SKPC, on the other hand, finds the first character in the string that is *different* from the search character you supply. This is useful for skipping through fill characters at the end of a field to find the beginning of the next field.

The Match Characters (MATCHC) instruction is similar to the Locate Character instruction, but it locates multiple-character substrings. MATCHC searches a string for the first occurrence of a substring you supply.

The Span Characters (SPANC) and Scan Characters (SCANC) instructions are search instructions that look for members of character classes. For these instructions you supply a character string, a mask, and the address of a 256-byte table of character type definitions. For each character in the given string, the instruction looks up the type code in the table for that character, and then AND's the given mask with the character's type code. SPANC finds the first character in the string which is of the type indicated by its mask. SCANC finds the first character in the string which is of any type other the one indicated by its mask.

The Index Instruction

The Index instruction (INDEX) calculates an index for an array of fixed length data types (integer and floating) and for arrays of bit fields, character strings, and decimal strings. It accepts as arguments: a subscript, lower and upper subscript bounds, an array element size, a given index, and a destination for the calculated index. It incorporates range checking within the calculation for high-level languages using subscript bounds, and it allows index calculation optimization by removing invariant expressions.

The COBOL statements:

```
01  A-ARRAY
    02      A PIC X(10) OCCURS 15 TIMES.
01  B PIC X(10).
    MOVE A(I) TO B.
```

are equivalent to:

```
INDEX I, #1, #15, #10, #0, R0   ; 1 less than or equal I
                                ; I less than or equal 15
                                ; (0 + I) * 10 is
                                ; stored in R0
MOVC3 #10, A-10[R0],B
```

The FORTRAN statements:

```
INTEGER*4 A(L1:U1, L2:U2), I, J
A(I,J) = 1
```

are equivalent to:

```
INDEX J, #L2, #U2, #M1, #0, R0      ; M1 = U1 - L1
INDEX I, #L1, #U1, #1, R0, R0
MOVL    #1, A-a[R0]                 ; a = ((L2*M1)+L1)*4
```

Variable-Length Bit Field Instructions

The bit field instructions enable you to define, access, and modify fields whose size and location you specify. Location is determined from a base address or a register and a signed bit offset. If the field is in memory, the offset range can be as large as 2^{32}-1 bits (approximately 16 million bytes). If the field is in a register, the offset can be large as 31. You can use fields of arbitrary lengths (0 to 32 bits) for storing data structure header information compactly, for status codes, or for creating your own data types. The field instructions enable you to manipulate fields easily.

The Insert Field and Extract Field instructions store data in and retrieve data from fields. Insert Field (INSV) stores data in a field by taking a specified number of bits of a longword (starting from the low-order bit) and writing them into a field, which may start at any bit relative to a given base address. The Extract Field instruction retrieves data from a field by copying the bit field and storing it in the low-order bits of a longword. The field can either be signed (EXTV) or unsigned (EXTZV).

The Compare Field and Find First instructions enable you to test the contents of a field. Compare Field extracts a field and then compares it with a given longword. The field can be interpreted as signed (CMPV), or as unsigned (CMPZV). The Find First instructions locate the first bit in a field that is clear (FFC) or set (FFS), scanning from low-order bit to high-order bit. These instructions are particularly useful for scanning a status control longword. For example, the longword may represent a set of queues processed in order by priority 0 (high) to 31 (low). Each set bit represents an active queue. The Find First Set instruction quickly returns the highest priority queue that is active. Together with the SKPC instructions, the Find First instructions are also useful for scanning an allocation table (bit map) of arbitrary length.

Queue Instructions

The processor has two instructions that enable you to construct and maintain queue data structures easily. Queues manipulated using the queue instructions are circular, doubly linked lists of data items.

The first longword of a queue entry contains the forward pointer to the next entry in the queue, and the next longword contains the backward pointer to the preceding entry in the queue. One queue entry is arbitrarily treated as the head of the queue. Since a list is circular, the tail of a queue is the entry that points to the head of the queue. In practice, the first entry of a queue is a "permanently allocated" listhead containing only the pointers to the first and last elements.

The INSQUE instruction inserts entries into queues. If an entry is the first item in a queue, INSQUE effectively creates a queue. The REMQUE instruction removes entries from a queue, and effectively deletes a queue if an entry is the last item removed. Entries can be inserted or removed at the head or tail of a queue, or anywhere in between.

Cooperating processes can access the same queue at the same time without external synchronization. If you allow more than one process to access a given queue at the same time, however, they should insert or remove entries only at the head or the tail of the queue. If you allow only one process at a time to access a queue, it can insert or remove entries anywhere in the queue.

Furthermore, because INSQUE and REMQUE are non-interruptable instructions, the operating system's procedures executing in the context of a process can share queues with interrupt service routines.

Address Manipulation Instructions

Because the processor offers a variety of addressing modes that enable you to access data structures easily by keeping base addresses and indices in registers, you often manipulate addresses. The processor provides two instructions that enable you to fetch an address without actually accessing the data at that location:

- the Move Address (MOVA) instruction, which stores the address of a byte, word, longword (and floating), or quadword (and double floating) datum in a specified register or location in memory.
- the Push Address (PUSHA) instruction, which stores the address of a byte, word, longword (and floating), or quadword (and double floating) datum on the stack.

Push Address is useful for computing an address that you want to pass to a subroutine or procedure you call. Move Address is useful for loading a base register and performing run-time position independent address computation. It has some interesting uses because it is effectively an ADD instruction:

```
MOVAB  disp(R1)[R2],X   ; sets X = R1+R2+displacement
                        ; (two adds in one instruction)

MOVA  disp(Rn)[Rn],Rn   ; multiplies Rn by
                        ; 3 (for MOVAW)
                        ; 5 (for MOVAL), or
                        ; 9 (for MOVAQ)
                        ; and adds displacement to it.
```

General Register Manipulation Instructions

The general register manipulation instructions enable any user program to save or load the general purpose registers in one operation, examine the Processor Status Longword, and set or clear status bits in the Processor Status Word. (Processor register control instructions primarily used by operating system software are covered later.)

The Push Longword (PUSHL) instruction pushes a longword on the stack. This instruction is the same as a Move Longword using the Stack Pointer in register deferred mode, but is a byte shorter. It is a consistent and convenient way to move data to the stack.

The Push Registers (PUSHR) instruction pushes a set of registers on the stack in one operation. You supply a mask word in which each set bit (0-14) represents a register (R0 through R14) that you want saved on the stack. (The only general register you cannot save using this instruction is R15, the Program Counter.) Pop Registers (POPR) reverses the operation, loading each register from successive longwords on the stack according to the given mask word. The PUSHR and POPR instructions replace the need to write a sequence of Move instructions to save and restore registers upon entry and exit from a subroutine.

The Move from Processor Status Longword (MOVPSL) instruction allows you to examine the contents of the processor's status register by loading its contents into a specified location. The Bit Set (BISPSW) and Bit Clear (BICPSW) Processor Status Word instructions allow you to set or clear the PSW condition codes and trap enable bits. You supply a mask in which each bit represents the bits you want set or cleared.

Branch, Jump and Case Instructions

The two basic types of control transfer instructions are branch and jump instructions. Both branch and jump load new addresses in the Program Counter. With branch instructions, you supply a displacement (offset) which is added to the current contents of the Program Counter to obtain the new address. With jump instructions, you supply the address you want loaded, using one of the normal addressing modes.

Because most transfers are to locations relatively close to the current instruction, and branch instructions are more efficient than jump instructions, the processor offers a variety of branch instructions to choose from. There are two unconditional branch instructions and many conditional branch instructions.

The unconditional branch instructions allow you to specify a byte-size (BRB) or a word-size displacement (BRW), which means you can branch to locations as far away from the current location as 32,767 bytes in either direction. For control transfers to locations farther away, you can use the Jump instruction (JMP).

Most conditional branches allow only byte displacements, although some of the more powerful, such as the Add Compare and Branch instruction, allow word displacements. Conditional branch instructions include:

- branch on bit instructions
- set and clear bit instructions with a branch if it is already set or cleared
- loop instructions that increment or decrement a counter, compare it with a limit value, and branch on a relational condition
- computed branch instruction in which a branch may take place to one of several locations depending on a computed value

The Branch on Condition (B) instructions enable you to transfer control to another location depending on the status of one or more of the condition codes in the Processor Status Word (PSW). There are three groups of Branch on Condition instructions:

- the signed relational branches, which are used to test the outcome of instructions operating on integer and field data types being treated as signed integers, floating point data types, and decimal strings
- the unsigned relational branches, which are used to test the outcome of instructions operating on integer and field data types being treated as unsigned integers, character strings, and addresses
- the overflow and carry test branches, which are used for checking overflow when traps are not enabled, for multiprecision arithmetic, and for the results of special instructions

The instruction mnemonics clearly indicate the choice between a signed and unsigned integer data type interpretation for relational testing. The relational tests enable you to determine if the result of the previous operation is less than, less than or equal, equal, not equal, greater than or equal, or greater than zero. For example, the Branch on Less than or Equal Unsigned (BLEQU) instruction branches if either the Carry or Zero bit is set. The Branch on Greater Than (BGTR) instruction branches if neither the Negative nor the Zero bit is set.

There are also general purpose Branch on Bit instructions similar to Branch on Condition. The Branch on Low Bit Set (BLBS) and Branch on Low Bit Clear (BLBC) instructions test bit 0 of an operand, which is useful for testing Boolean values. The Branch on Bit Set (BBS) and Branch on Bit Clear (BBC) instructions test any selected bit.

There are special kinds of Branch on Bit instructions that are actually bit set/clear instructions. The Branch on Bit Set and Set (BBSS) is an example. The instruction branches if the indicated bit is set, otherwise it falls through. In either case, the instruction sets the given bit. The BBSS instruction can thus be thought of as a Bit Set instruction with a branch side-effect if the bit was already set. There are four permutations:

- Branch on Bit Set and Set (BBSS)
- Branch on Bit Clear and Clear (BBCC)
- Branch on Bit Set and Clear (BBSC)
- Branch on Bit Clear and Set (BBCS)

These instructions are particularly useful for keeping track of procedure completion or initialization, and for signaling the completion or initialization of a procedure to a cooperating process. In addition, there are two Branch on Bit Interlocked instructions that provide control variable protection:

- Branch on Bit Set and Set Interlocked (BBSSI)
- Branch on Bit Clear and Clear Interlocked (BBCCI)

The SBI bus provides a memory interlock on these instructions. No other BBSSI or BBCCI operation can interrupt these instructions to gain access to the byte containing the control variable between the testing of the bit and the setting or clearing of the bit.

The processor offers three types of branch instruction that can be used to write efficient loops. The first type provides the basic subtract-one-and-branch loop. You supply a counter variable which is decremented each time the loop is executed. In the Subtract One and Branch Greater Than (SOBGTR) instruction, the loop repeats until the counter equals zero. In the Subtract One and Branch Greater Than or Equal (SOBGEQ) instruction, the loop repeats until the counter becomes negative.

The counterpart to subtract-one-and-branch is add-one-and-branch. In this case you supply a counter and a limit. The counter is incremented at the end of the loop. In the Add One and Branch Less Than (AOBLSS) instruction, the loop repeats until the counter equals the limit you set. In the Add One and Branch Less Than or Equal (AOBLEQ) instruction, the loop repeats until the counter exceeds the limit you set.

The third type of loop instruction efficiently implements the FORTRAN language DO statement and the BASIC language FOR statement: Add Compare and Branch (ACB). You supply a limit, a counter, and a step value. For each execution of the loop, the instruction adds the step value to the counter and compares the counter to the limit. The sign of the step value determines the logical relation of the comparison: the instruction loops on a less than or equal comparison if the step value is positive, on a greater than or equal comparison if the step value is negative.

The processor provides a branch instruction that implements higher-level language computed GO TO statements: the CASE instruction. For CASE, you supply a list of displacements that generate different branch addresses indexed by the value you obtain as a selector. The branch falls through if the selector does not fall within the limits of the list.

Subroutine Branch, Jump, and Return Instructions

Two special types of branch and jump instruction are provided for calling subroutines: the Branch to Subroutine (BSB) and Jump to Subroutine (JSB) instructions. Both BSB and JSB instructions save the contents of the Program Counter on the stack before loading the Program Counter with the new address. With Branch to Subroutine, you can supply either a byte (BSBB) or word (BSBW) displacement. With Jump to Subroutine, regular addressing is used.

The subroutine call instructions are complemented by the Return from Subroutine (RSB) instruction. RSB pops the first longword off the stack and loads it into the Program Counter. Since the Branch to Subroutine instruction is either two or three bytes long, and the Return from Subroutine instruction is one byte long, it is possible to write extremely efficient programs using subroutines.

Procedure Call and Return Instructions

Procedures are general purpose routines that use argument lists passed automatically by the processor. The procedure Call instructions enable language processors and the operating system to provide a standard calling interface. They:

- save all the registers that the procedure uses, and only those registers, before entering the procedure
- pass an argument list to a procedure

- maintain the Stack, Frame, and Argument Pointer registers
- initialize the arithmetic trap enables to a given state

When you issue a Call procedure instruction, you supply the address of the procedure you are calling. The first word of a procedure contains an entry mask that is used in the same way as the entry mask defined for the Push Registers instruction. Each set bit of the 12 low-order bits in the word represents one of the general registers, R0 through R11, that the procedure uses. The Call instruction examines this word and saves the indicated registers on the stack. In addition, the Call instruction also automatically saves the contents of the Frame Pointer, Argument Pointer, and Program Counter registers. This is an extremely efficient way to ensure that registers are saved across procedure calls. No general register is saved that does not have to be saved.

The Call Procedure with General Argument List (CALLG) instruction accepts the address of an argument list and passes the address to the procedure in the Argument Pointer register. The Call Procedure with Stack Argument List (CALLS) passes the argument list, if any, which you have placed on the stack, by loading the Argument Pointer register with its stack address.

When a procedure completes execution, it issues the Return from Procedure instruction (RET). Return uses the Frame Pointer register to find the saved registers that it restores, and to clean up any data left on the stack, including nested routine linkages. A procedure can return values using the argument list or other registers.

Miscellaneous Special Purpose Instructions

The processor has a number of special purpose instructions. They include:

- Cyclic Redundancy Check (CRC)
- Breakpoint Fault (BPT)
- Extended Function Call (XFC)
- No Operation (NOP)
- Halt

The Cyclic Redundancy Check (CRC) instruction calculates a cyclic redundancy check for a given string using any CRC polynomial up to 32 bits long. You supply the string for which you want the CRC, and a table for the CRC function. The operating system library includes tables for standard CRC functions, such as CRC-16.

The Breakpoint Fault (BPT) instruction makes the processor execute the kernel mode condition handler associated with the Breakpoint Fault exception vector. BPT is used by the operating system debugging utilities, but can also be used by any process that sets up a Breakpoint Fault condition handler.

The Extended Function Call (XFC) instruction allows escapes to customer-defined instructions in writable control store. The NOP instruction is useful for debugging. The HALT instruction is a privileged instruction issued only by the operating system to halt the processor when bringing the system down by operator request.

COMPATIBILITY MODE

Under control of the operating system, the processor can execute PDP-11 instruction streams within the context of any process. When executing in compatibility mode, the processor interprets the instruction stream executing in the context of the current process as a subset of PDP-11 code that does not include floating point hardware instructions or privileged instructions.

In general, compatibility mode enables the operating system to provide an environment for executing most user mode programs written for a PDP-11 except stand-alone software. The processor expects all compatibility mode software to rely on the services of the native operating system for I/O processing, interrupt and exception handling, and memory management. There are some restrictions, however, on the environment that the native operating system can provide a PDP-11 program. For example, the PDP-11 memory management instructions Move To/From Previous Instruction/Data Space can not be simulated by the operating system since they do not trap to native mode software.

PDP-11 Program Environment

PDP-11 addresses are 16-bit byte addresses. There is a one-to-one correspondence between compatibility mode virtual addresses and the first 64K bytes of virtual address space available to native mode processes. As in the PDP-11, a compatibility mode program is restricted to referencing only these addresses. It is possible for the operating system to provide most of the PDP-11 memory management mechanisms. For example, compatibility mode automatically supports PDP-11 memory segment protection, but in 512-byte rather than 64-byte segments.

All of the PDP-11 general registers and addressing modes are available in compatibility mode. Compatibility mode registers R0 through R6 are the low-order 16 bits of native mode registers R0 through R6. Compatibility mode R7 (the Program Counter) is the low-order bits of native mode register 15 (the Program Counter). Native mode registers 8 through 14 are not affected by compatibility mode. Note that the compatibility mode register R6 acts as the Stack Pointer for program-local temporary data storage, but that the program-local stack is allocated address space in the program region, not the control region.

A subset of the PDP-11 Processor Status Word is defined for compatibility mode. Only the condition codes and the trace trap bit are relevant for the PDP-11 instruction stream.

All interrupts and exceptions that occur when the processor is executing in compatibility mode cause the processor to enter native mode. As in native mode, it is the operating system's responsibility to handle interrupts and exceptions. There are a few types of exceptions that apply only to compatibility mode. They include illegal instruction exceptions and odd address trap.

PDP-11 Instruction Set

The compatibility mode instruction set is that of the PDP-11 with the following exceptions:

- the privileged and floating point option instructions are illegal (this includes HALT, WAIT, RESET, SPL, MARK, the floating instruction set, and the floating point processor instructions)

- the trap instructions (BPT, IOT EMT, and TRAP) cause the processor to enter native mode, where either the trap may be serviced, or the instruction simulated
- the move from/to previous instruction/data space instructions (MFPI, MTPI, MFPD, and MTPD) execute exactly as they would on a PDP-11 in user mode with instruction and data space overmapped. They ignore the previous access level and act as PUSH and POP instructions referencing the current stack.

All other instructions execute exactly as they would on a PDP-11/70 processor running in user mode.

PROCESSING CONCEPTS FOR SYSTEM PROGRAMMING

The processor is specifically designed to support a high-performance multiprogramming environment. The chief advantage of a multiprogramming system is its ability to get the most out of a computer that is being used for several different purposes concurrently. For example, multiprogramming enables you to run two or more application systems, such as order entry and customer billing, at the same time. Or, you can be running one or more application systems while you are developing application programs. The characteristics of the hardware system that support multiprogramming are:

- rapid context switching
- priority dispatching
- virtual addressing and memory management

As a multiprogramming system, VAX-11/780 not only provides the ability to share the processor among processes, but also protects processes from one another while enabling them to communicate with each other and share code and data.

Context Switching

In a multiprogramming environment, several individual streams of code can be ready to execute at any one time. Instead of allowing each stream to execute to completion serially (as in a batch-only system), the operating system can intervene and switch between the streams of code which are ready to execute.

To support multiprogramming for a high-performance system, the processor enables the operating system to switch rapidly between individual streams of code. The stream of code the processor is executing at any one time is determined by its *hardware context*. Hardware context includes the information loaded in the processor's registers that identifies:

- where the stream's instructions and data are located
- which instruction to execute next
- what the processor status is during execution

A process is a stream of instructions and data defined by a hardware context. Each process has a unique identification in the system. The operating system switches between processes by requesting the processor to save one process hardware context and load another. Context switching occurs rapidly because the processor instruction set includes save hardware context and load hardware context instructions. The operating system's context switching software does not have to individually save or load the processor registers which define the hardware context.

The actual scheduling mechanism for arbitrating among processes competing for processor time is left to the operating system software itself to give the system flexibility.

Priority Dispatching

While running in the context of one process, the processor executes instructions and controls data flow to and from peripherals and main memory. To share processor, memory and peripheral resources among many processes, the processor provides two arbitration mechanisms that support high-performance multiprogramming: exceptions and interrupts. Exceptions are events that occur synchronously with respect to instruction execution, while interrupts are external events that occur asynchronously.

The flow of execution can change at any time, and the processor distinguishes between changes in flow that are local to a process and those that are system-wide. Process-local changes occur as the result of a user software error or when user software calls operating system services. Process-local changes in program flow are handled through the processor's exception detection mechanism and the operating system's exception dispatcher.

System-wide changes in flow generally occur as the result of interrupts from devices or interrupts generated by the operating system software. Interrupts are handled by the processor's interrupt detection mechanism and the operating system's interrupt service routines. (System-wide changes in flow may also occur as the result of severe hardware errors, in which case they are handled either as special exceptions or high-priority interrupts.)

System-wide changes in flow take priority over process-local changes in flow. Furthermore, the processor uses a priority system for servicing interrupts. To arbitrate between all possible interrupts, each kind of interrupt is assigned a priority, and the processor responds to the highest priority interrupt pending. For example, interrupts from the high-speed disk devices take precedence over interrupts from low-speed devices.

The processor services interrupts between instructions, or at well-defined points during the execution of long, iterative instructions. When the processor acknowledges an interrupt, it switches rapidly to a special system-wide context to enable the operating system to service the interrupt. System-wide changes in the flow of execution are handled in such a way as to be totally transparent to individual processes.

Virtual Addressing and Virtual Memory

The processor's memory management hardware enables the operating system to provide an execution environment that allows users to write programs without having to know where the programs are loaded in physical memory, and to write programs that are too large to fit in the physical memory they are allocated.

The processor provides the operating system with the ability to provide virtual addressing. A virtual address is a 32-bit integer that a program uses to identify storage locations in virtual memory. Virtual memory is tne set of all physical memory locations in the system plus the set of disk blocks that the operating system designates as extensions to physical memory.

A physical address is an address that the processor uses to identify physical memory storage locations and peripheral controller registers. It is the address that the processor sends out on the SBI bus to which the memory and peripheral adaptors respond.

The processor must be able to translate the virtual addresses provided by the programs it executes into the physical addresses recognized by the memory and peripherals. To provide virtual to physical address mapping, the processor has address mapping registers controlled by the operating system and an integrated address translation buffer.

The mapping registers enable the operating system to relocate programs in physical memory, to protect programs from each other, and share instructions and data between programs transparently or at their request. The address translation buffer ensures that the virtual address to physical address translation takes place rapidly.

SYSTEM PROGRAMMING ENVIRONMENT

Within the context of one process, user-level software controls its execution using the instruction sets, the general registers and the Processor Status Word. Within the multiprogramming environment, the operating system controls the system's execution using a set of special instructions, the Processor Status Longword, and the internal processor registers.

Processor Status Longword

A processor register called the Processor Status Longword (PSL) determines the execution state of the processor at any time. The low-order 16 bits of the Processor Status Longword is the Processor Status Word available to the user process. The high-order 16 bits provide privileged control of the system. Figure 4-6 illustrates the Processor Status Longword.

The fields can be grouped together by functions that control:

- the instruction set the processor is executing
- the access mode of the current instruction
- interrupt processing

The instruction set the processor executes is controlled by the compatibility mode bit in the Processor Status Longword. This bit is normally set or cleared by the operating system. For further information on compatibility mode, refer to the Operating System section.

The following paragraphs discuss access modes, the native instructions primarily used by the operating system, memory management, and interrupt processing.

Processor Access Modes

In a high-performance multiprogramming system, the processor must provide the basis for protection and sharing among the processes competing for the system's resources. The basis for protection in this system is the processor's access mode. The access mode in which the processor executes determines:

- instruction execution privileges: what instructions the processor will execute
- memory access privileges: which locations in memory the current instruction can access

At any one time, the processor is executing code in the context of a particular process, or it is executing in the system-wide interrupt service context. In the context of a process, the processor recognizes four access modes: kernel, executive, supervisor, and user. Kernel is the most privileged mode and user the least.

The processor spends most of its time executing in user mode in the context of one process or another. When user software needs the services of the operating system, whether for acquisition of a resource, for I/O processing, or for information, it calls those services.

The processor executes those services in the same or one of the more privileged access modes within the context of that process. That is, all four access modes exist within the same virtual address space. Each access mode has its own stack in the control region of per-process space, and therefore each process has four stacks: one for each access mode. Note that this makes it easy for the operating system to context switch a process even when it is executing an operating system service procedure.

In any mode except kernel, the processor will not execute the instructions that:

- halt the processor
- load and save process context
- access the internal processor registers that control memory management, interrupt processing, the processor console, or the processor clock

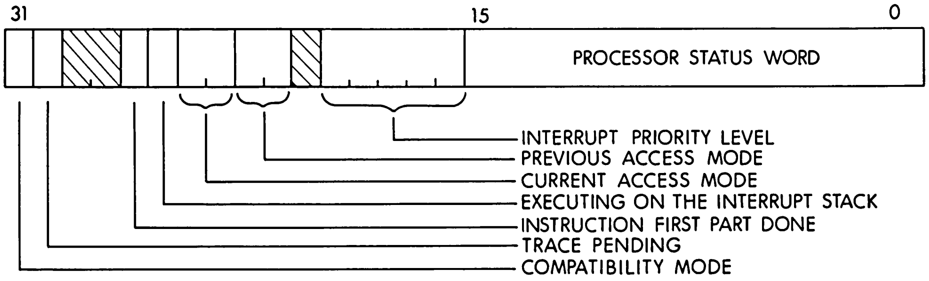


Figure 4-6
Processor Status Longword

These instructions are privileged instructions that are generally reserved to the operating system.

In any mode, the processor will not allow the current instruction to access memory unless the mode is privileged to do so. The ability to execute code in one of the more privileged modes is granted by the system manager and controlled by the operating system. The memory protection the privilege affords is enforced by the processor. In general, code executing in one mode can protect itself and any portion of its data structures from read and/or write access by code executing in any less privileged mode. For example, code executing in executive mode can protect its data structures from code executing in supervisor or user mode. Code executing in supervisor mode can protect its data structures from access by code executing in user mode. This memory protection mechanism provides the basis for system data structure integrity.

Protected and Privileged Instructions

The processor provides three types of instructions that enable user mode software to obtain operating system services without jeopardizing the integrity of the system. They include:

- the Change Mode instructions
- the PROBE instructions
- the Return from Exception or Interrupt instruction

User mode software can obtain privileged services by calling operating system service procedures with a standard CALL instruction. The operating system's service dispatcher issues an appropriate Change Mode instruction before actually entering the procedure. Change Mode allows access mode transitions to take place from one mode to the same or more privileged mode only. When the mode transition takes place the previous mode is saved in the Previous Mode field of the Processor Status Longword, allowing the more privileged code to determine the privilege of its caller.

A Change Mode instruction is simply a special trap instruction that can be thought of as an operating system service call instruction. User mode software can explicitly issue Change Mode instructions, but since the operating system receives the trap, non-privileged users can not write any code to execute in any of the privileged access modes. User mode software can include a condition handler for Change Mode to User traps, however, and this instruction is useful for providing general purpose services for user mode software. The system manager ultimately grants the privilege to write any code that handles Change Mode traps to more privileged access modes.

For service procedures written to execute in privileged access modes (kernel, executive, and supervisor), the processor provides address access privilege validation instructions. The PROBE instructions enable a procedure to check the read (PROBER) and write (PROBEW) access protection of pages in memory against the privileges of the caller who requested to access a particular location. This enables the operating system to provide services that execute in privileged modes to less privileged callers and still prevent the caller from accessing protected areas of memory.

The operating system's privileged service procedures and interrupt and exception service routines exit using the Return from Exception or Interrupt (REI) instruction. REI is the only way in which the privilege of the processor's access mode can be decreased. Like the procedure and subroutine return instructions, REI restores the Program Counter and the processor state to resume the process at the point where it was interrupted.

REI performs special services, however, that normal return instructions do not. For example, REI checks to see if any asynchronous system traps have been queued for the currently executing process while the interrupt or exception service routine was executing, and ensures that the process will receive them. Furthermore, REI checks to ensure that the mode to which it is returning control is the same as or less privileged than the mode in which the processor was executing when the exception or interrupt occurred. Thus REI is available to all software including user-written trap handling routines, but a program can not increase its privilege by altering the processor state to be restored.

When the operating system schedules a context switching operation, the context switching procedure uses the Save Process Context (SVPCTX) and Load Process Context (LDPCTX) instructions to save the current process context and load another. The operating system's context switching procedure identifies the location of the hardware context to be loaded by updating an internal processor register.

Internal processor registers not only include those that identify the process currently executing, but also the memory management and other registers, such as the console and clock control registers. The Move to Processor Register (MTPR) and Move from Processor Register (MFPR) instructions are the only instructions that can explicitly access the internal processor registers. MTPR and MFPR are privileged instructions that can be issued only in kernel mode.

Memory Management

The processor is responsible for enforcing memory protection between access modes. Memory protection, however, is only a part of the processor's memory management function. In particular, the memory management hardware enables the operating system to provide an extremely flexible and efficient virtual memory programming environment. Virtual and physical address space definitions provide the basis for the virtual memory available on a system. Figure 4-7 compares the structures of the two address spaces.

Virtual address space consists of all possible 32-bit addresses that can be exchanged between a program and the processor to identify a byte location in physical memory. The memory management hardware translates a virtual address into a 30-bit physical address. A physical address is the address exchanged between the processor and the memory and peripheral adaptors over the SBI bus. Physical address space is the set of all possible physical addresses the processor can use to express unique memory locations and peripheral control registers.

Physical address space is an array of addresses which can be used to represent 2^{30} byte locations, or approximately one billion bytes. Half of the addresses in physical address space can be used to refer to real memory locations and the other half can be used to refer to peripheral device control and data registers. The lowest-addressed half of physical address space is called **memory space**, and the highest-addressed half **I/O space** (discussed later on).

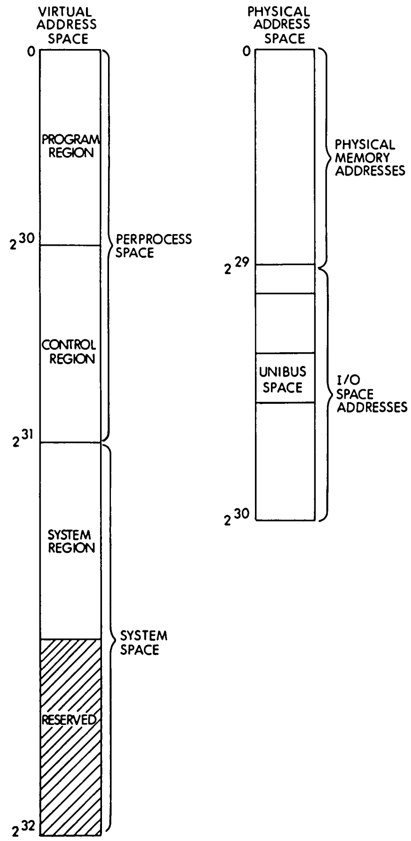


Figure 4-7
Virtual and Physical Address Space

The following section describes the way in which the memory management hardware enables the operating system to map virtual addresses into physical addresses to provide the virtual memory available to a process.

Virtual to Physical Page Mapping

Virtual address space is divided into pages, where a page represents 512 bytes of contiguously addressed memory. The first page begins at byte zero and continues to byte 511. The next page begins at byte 512 and continues to byte 1023, and so forth. If we listed the first 8 pages of virtual address space, their addresses in both decimal and hexadecimal are:

PAGE	ADDRESS(10)	ADDRESS(16)
0	0000-0511	0000-01FF
1	0512-1023	0200-03FF
2	1024-1535	0400-05FF
3	1536-2047	0600-07FF
4	2048-2559	0800-09FF
5	2560-3071	0A00-0BFF
6	3072-3583	0C00-0DFF
7	3584-4095	0E00-0FFF

The size of a virtual page exactly corresponds to the size of a physical page of memory, and the size of a block on disk.

To make memory mapping efficient, the processor must be able to translate virtual addresses to physical addresses rapidly. Two features providing rapid address translation are the processor's internal address translation buffer, which is described later, and the translation algorithm itself.

Figure 4-8 compares the virtual and physical address format. The high-order two bits of a virtual address immediately identify the region to which the virtual address refers. Whether the address is physical or virtual, the byte within the page is the same. Thus, the processor has to know only which virtual pages correspond to which physical pages.

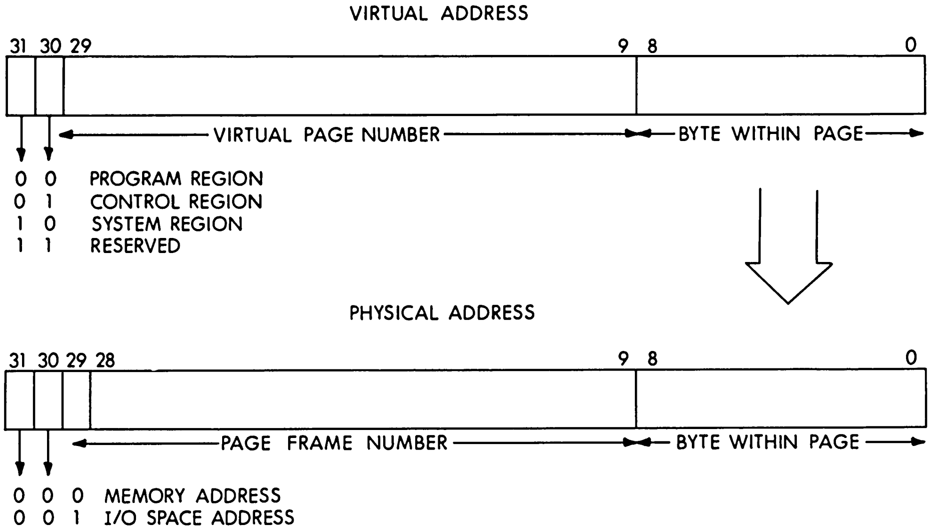


Figure 4-8
Virtual and Physical Addresses

The processor has three pairs of page mapping registers, one pair for each of the three regions actively used. The operating system's memory management software loads each pair of registers with the base address and length of data structures it sets up called **page tables**. The page tables provide the mapping information for each virtual page in the system. There is one page table for each of the three regions.

A page table is a virtually contiguous array of page table entries. Each page table entry is a longword representing the physical mapping for one virtual page. To translate a virtual address to a physical address, therefore, the processor simply uses the virtual page number as an index into the page table from the given page table base address. Each translation is good for 512 virtual addresses since the byte within the virtual page corresponds to the byte within the physical page.

Figure 4-9 shows the format of a page table entry. The high-order bits are used to indicate the page's status and protection. The page's protection can be set to prevent read and/or write access by any mode (kernel, executive, supervisor, or user). The page's status indicates what the remainder of the page table entry means. It may be, for example, a page address in physical address space, a disk sector, or a temporary pointer to a page shared by two or more processes. The system's **virtual memory** is a dynamic memory that is defined by the physical memory and disk pages that are virtually mapped by page table entries.

The operating system's memory management software maintains the page table entry protection and status bits, with the exception of the modified page bit. The processor sets the modified page bit to indicate that it has written into a physical page in memory. This is used to keep disk I/O to a minimum when paging a process.

The processor uses the page table base registers to locate the page tables, and uses the length registers as a validity check to ensure that any given virtual page is in the range of defined page table entries. Figure 4-10 summarizes and compares the page table structures.

All process page tables have virtual addresses in the system region of virtual address space, but the system region page table is located by its address in physical memory. That is, the system region page table base register contains the *physical address* of the page table base, while the process page table base registers contain the *virtual addresses* of their page table bases. Because a per-process page table entry is referred to by its virtual address, the hardware translates its virtual address using the system page table.

There are two advantages to using a virtual address as the base address of a per-process page table. The first advantage is that all page tables do not have to reside in physical memory. The system region page table is the only page table that needs to be resident in physical memory. All process page tables can reside on disk; that is, process page tables can themselves be paged and swapped as necessary.

The second advantage is that the operating system's memory management software can allocate per-process page tables dynamically, because the per-process page tables do not need to be mapped into contiguous physical pages. And although the system region page table must be mapped into contiguous physical pages, this requirement does not restrict physical memory allocation. The region is shared among processes, and therefore does not require redefinition from context to context.

To illustrate the efficiency of this memory mapping scheme, suppose that 16 processes, each of which is us-

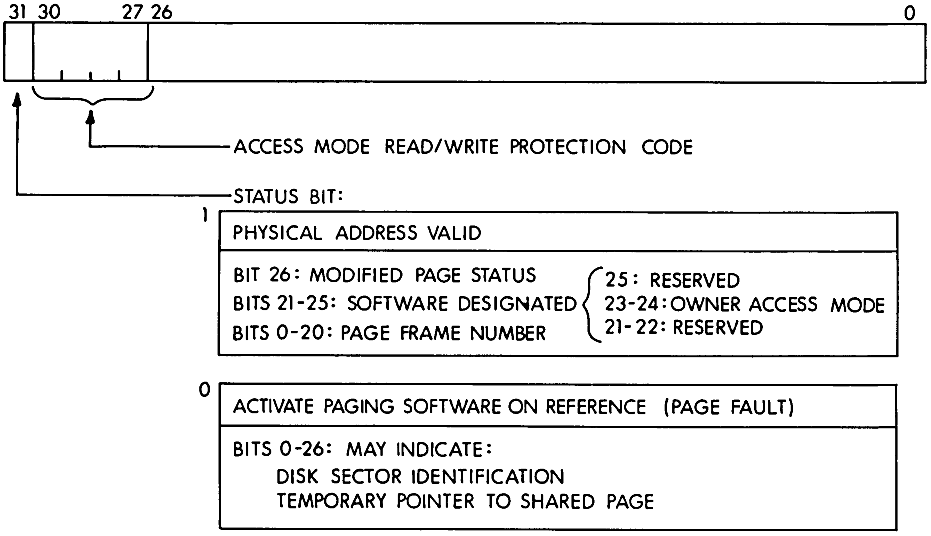

31 30 27 26 0
ACCESS MODE READ/WRITE PROTECTION CODE
STATUS BIT:
1
PHYSICAL ADDRESS VALID
BIT 26: MODIFIED PAGE STATUS
BITS 21-25: SOFTWARE DESIGNATED
BITS 0-20: PAGE FRAME NUMBER
25: RESERVED
23-24: OWNER ACCESS MODE
21-22: RESERVED
0
ACTIVATE PAGING SOFTWARE ON REFERENCE (PAGE FAULT)
BITS 0-26: MAY INDICATE:
DISK SECTOR IDENTIFICATION
TEMPORARY POINTER TO SHARED PAGE

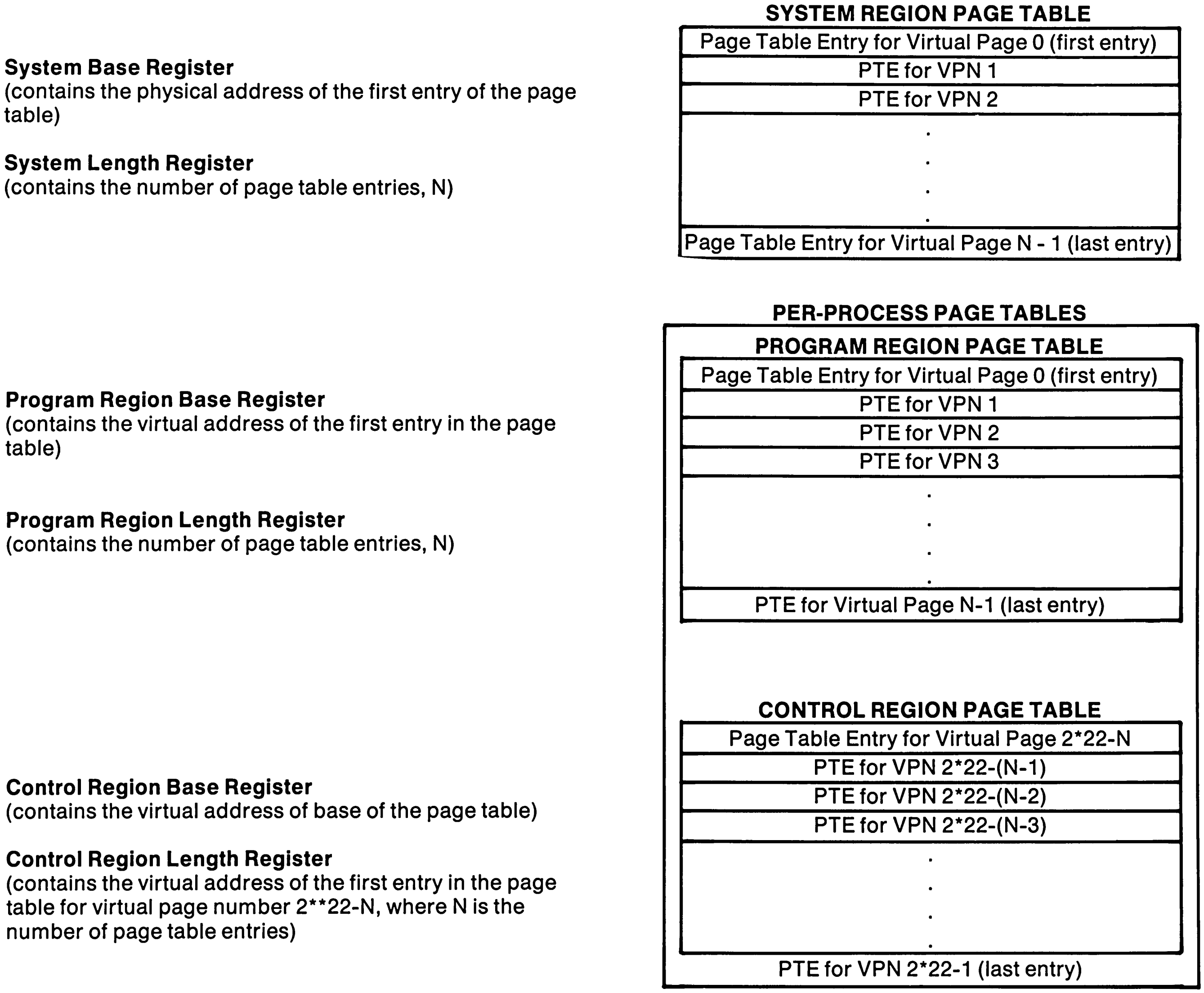
Figure 4-9
Page Table Entry

ing 4 million bytes of virtual address space, are known to the system at the same time (for a total of 64MB of virtual address space). One system page table entry maps one page of per-process page table entries, and one page of per-process page table entries maps 65,536 (64K) bytes of virtual address space (since it is possible to store 128 page table entries in a single page of memory). Therefore one page of system page table maps 128 pages of per-process page tables, which in turn maps 8MB of process virtual address space. Thus the system region page table needed to map these 16 processes requires approximately 8 physical pages (4K bytes) of memory.

Exception and Interrupt Vectors

The processor can automatically initiate changes in the normal flow of program execution. The processor recognizes two kinds of events that cause it to invoke conditional software: exceptions and interrupts. Some exceptions affect an individual process only, such as arithmetic traps, while others affect the system as a whole, for example, machine check. Interrupts include both device interrupts, such as those signaling I/O completion, and software-requested interrupts, such as those signaling the need for a context switch operation.

The processor knows which software to invoke when an exception or interrupt occurs because it references specific locations, called vectors, to obtain the starting address of the exception or interrupt dispatcher. The processor has one internal register, the System Control Block Base Register, which the operating system loads with the physical address of the base of the System Control Block, which contains the exception and interrupt vectors. The processor locates each vector by using a specific offset into the System Control Block. Figure 4-11 illustrates the vectors in the System Control Block. Each vector tells the processor how to service the event, and contains the system region virtual address of the routine to execute. Note that vector 14 (hex) can be used as a trap to writable control store to execute user-defined instructions, and the vector contains information passed to microcode.



System Base Register
(contains the physical address of the first entry of the page table)

System Length Register
(contains the number of page table entries, N)

SYSTEM REGION PAGE TABLE

Page Table Entry for Virtual Page 0 (first entry)
PTE for VPN 1
PTE for VPN 2
. . . .
Page Table Entry for Virtual Page N - 1 (last entry)

Program Region Base Register
(contains the virtual address of the first entry in the page table)

Program Region Length Register
(contains the number of page table entries, N)

PER-PROCESS PAGE TABLES

PROGRAM REGION PAGE TABLE

Page Table Entry for Virtual Page 0 (first entry)
PTE for VPN 1
PTE for VPN 2
PTE for VPN 3
. . . .
PTE for Virtual Page N-1 (last entry)

Control Region Base Register
(contains the virtual address of base of the page table)

Control Region Length Register
(contains the virtual address of the first entry in the page table for virtual page number 2**22-N, where N is the number of page table entries)

CONTROL REGION PAGE TABLE

Page Table Entry for Virtual Page 2*22-N
PTE for VPN 2*22-(N-1)
PTE for VPN 2*22-(N-2)
PTE for VPN 2*22-(N-3)
. . . .
PTE for VPN 2*22-1 (last entry)

Figure 4-10
Page Tables

4	Machine Check	EXCEPTION VECTORS
8	Kernel Stack Not Valid	
C	Power Fail	
10	Reserved or Privileged Instruction	
14	Customer Reserved Instruction	
18	Reserved or Illegal Operand	
1C	Reserved or Illegal Addressing Mode	
20	Access Violation	
24	Translation Not Valid (page fault)	
28	Trace Trap	
2C	Breakpoint Trap	
30	Compatibility Mode Trap	
34	Arithmetic Trap	
⋮	⋮	
40	Change Mode to Kernel	
44	Change Mode to Executive	
48	Change Mode to Supervisor	
4C	Change Mode to User	
⋮	⋮	
84	Software Level 1	INTERRUPT VECTORS
88	Software Level 2	
BF	Software Level F	
CO	Interval Timer	
⋮	⋮	
100	Device Level 14, device 0	
101	Device Level 14, device 1	
⋮	⋮	
13F	Device Level 14, device 15	
140	Device Level 15, device 0	
⋮	⋮	
17F	Device Level 15, device 15	
180	Device Level 16, device 0	
⋮	⋮	
1BF	Device Level 16, device 15	
1C0	Device Level 17, device 0	
⋮	⋮	
1FF	Device Level 17, device 15	

Offset from System Control Block Base Register (HEX)

Figure 4-11
System Control Block

Table 4-3
Interrupt Priority Levels

PRIORITY		HARDWARE EVENT	
Hex	Decimal		
1F	31	Machine Check, Kernel Stack Not Valid	
1E	30	Power Fail	
1D	29		Processor, Memory, or Bus Error
1C	28		
1B	27		
1A	26		
19	25		
18	24	Clock	
17	23	UNIBUS BR7	Device Interrupt
16	22	UNIBUS BR6	
15	21	UNIBUS BR6	
14	20	UNIBUS BR4	
13	19		
12	18		
11	17		
10	16		
PRIORITY		SOFTWARE EVENT	
0F	15		Reserved for DIGITAL
0E	14		
0D	13		
0C	12		
0B	11		Device Drivers
0A	10		
09	09		
08	08		
07	07	Timer Process	
06	06	Queue Asynchronous System Trap (AST)	
05	05	Reserved for DIGITAL	
04	04	I/O Post	
03	03	Process Scheduler	
02	02	AST Delivery	
01	01	Reserved for DIGITAL	
00	00	User Process Level	

Interrupt Priority Levels

Exceptions do not require arbitration since they occur synchronously with respect to instruction execution. Interrupts, on the other hand, can occur at any time. To arbitrate between interrupt requests that may occur simultaneously, the processor recognizes 31 interrupt priority levels.

The highest 16 interrupt priority levels are reserved for interrupts generated by hardware, and the lowest 16 interrupt priority levels are reserved for interrupts requested by software. Table 4-3 lists the assignment of each level, from highest to lowest priority. Normal user software runs at process level, which is interrupt priority level zero.

To handle interrupt requests, the processor enters a special system-wide context. In the system-wide context, the processor executes in kernel mode using a special stack called the interrupt stack. The interrupt stack can not be referenced by any user mode software because the processor only selects the interrupt stack after an interrupt, and all interrupts are trapped through system vectors.

The interrupt service routine executes at the interrupt priority level of the interrupt request. When the processor receives an interrupt request at a level higher than that of the currently executing software, the processor honors the request and services the new interrupt at its priority level. When the interrupt service routine issues the REI (Return from Exception or Interrupt) instruction, the processor returns control to the previous level.

I/O Space and I/O Processing

An I/O device controller has a set of control/status and data registers. The registers are assigned addresses in physical address space, and their physical addresses are mapped, and thus protected, by the operating system's memory management software. That portion of physical address space in which device controller registers are located is called I/O space.

I/O space occupies the highest-addressed half of physical address space, and is 2^{29} bytes in length. A portion of I/O space is specifically mapped into UNIBUS addresses, and is called UNIBUS space.

No special processor instructions are needed to reference I/O space. The registers are simply treated as locations containing integer data. An I/O device driver issues commands to the peripheral controller by writing to the controller's registers as if they were physical memory locations. The software reads the registers to obtain the controller status. The driver controls interrupt enabling and disabling on the set of controllers for which it is responsible. When interrupts are enabled, an interrupt occurs when the controller requests it. The processor accepts the interrupt request and executes the driver's interrupt service routine if it is not currently executing on a higher-priority interrupt level.

Process Context

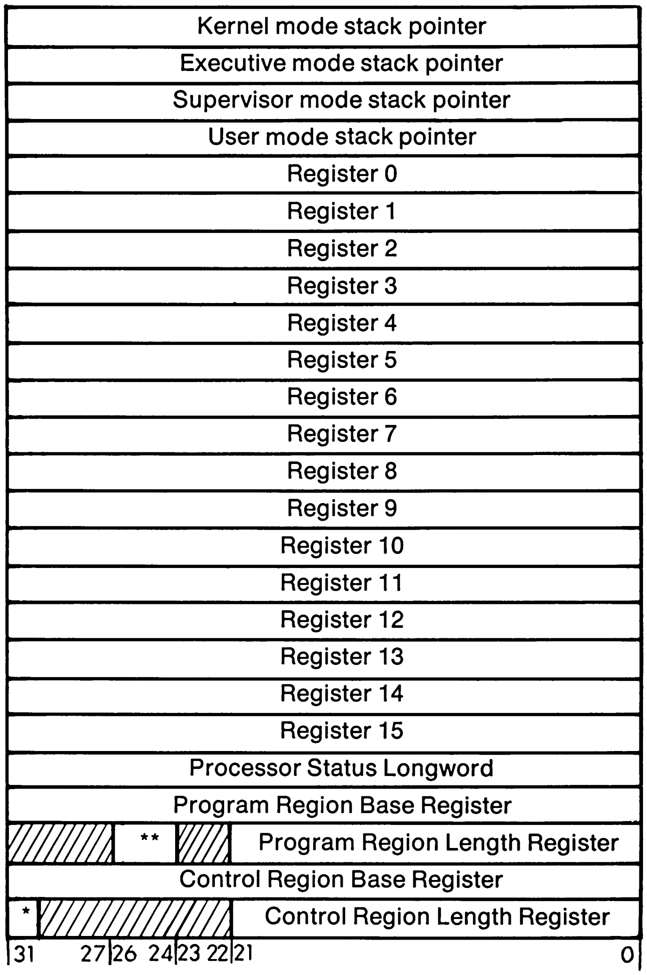
For each process eligible to execute, the operating system creates a data structure called the **software process control block**. Within the software process control block is a pointer to a data structure called the **hardware process control block**. The hardware process control block is illustrated in Figure 4-12. It contains the hardware process context, that is, all the data needed to load the processor's programmable registers when a context switch occurs. To give control of the processor to a process, the operating system loads the processor's Process Control Block Base register with the physical address of a hardware process control block and issues the Load Process Context instruction. The processor loads the process context in one operation and is ready to execute code within that context.

As can be seen from the illustration, a process control block not only contains the state of the programmable registers, it also contains the definition of the process virtual address space. Thus, the mapping of the process is automatically context-switched.

Furthermore, the process control block provides the mechanism for triggering asynchronous system traps to user processes. The Asynchronous System Trap field enables the processor to schedule a software interrupt to initiate an AST routine and ensure that they are delivered to the proper access mode for the process.

CONSOLE

The console is the operator's interface to the central processor. It consists of an LSI-11 processor with 16K bytes of read/write memory, 8K bytes of ROM, a floppy disk system, and a hard-copy terminal. Using the console terminal, the operator can examine and deposit data in memory locations or the processor registers, halt the processor, step through instruction streams, and boot the operating system.



Kernel mode stack pointer
Executive mode stack pointer
Supervisor mode stack pointer
User mode stack pointer
Register 0
Register 1
Register 2
Register 3
Register 4
Register 5
Register 6
Register 7
Register 8
Register 9
Register 10
Register 11
Register 12
Register 13
Register 14
Register 15
Processor Status Longword
Program Region Base Register
** Program Region Length Register
Control Region Base Register
* Control Region Length Register

*Enable performance monitor
**Asynchronous System Trap pending

Figure 4-12
Hardware Process Control Block

The console also serves as a microdiagnostic monitor. The floppy disk provides storage for basic diagnostic programs and data. The operator can load segments of diagnostic microcode from a floppy disk into the writable diagnostic control store, controlling execution and reporting results.

The console is further used for updating the software with maintenance releases and for loading optional software products distributed on floppy disk.

An EIA serial line interface and modem can be added to the console to provide remote diagnosis and automated testing.

FLOATING POINT ACCELERATOR

The floating point accelerator is an optional high-speed processor extension. When included in the processor, the floating point accelerator executes the addition, subtraction, multiplication, and division instructions that operate on single- and double-precision floating point operands, including the special EMOD and POLY instructions in both single- and double-precision formats. Additionally, the floating point accelerator enhances the performance of 32-bit integer multiply instructions.

The processor does not have to include the floating point accelerator to execute floating point operand instructions. The floating point accelerator can be added or removed without changing any existing software.

When the floating point accelerator is included in the processor, a floating point operand register-to-register add instruction takes as little as 800 nanoseconds to execute. A register-to-register multiply instruction takes as little as one microsecond. The inner loop of the POLY instruction takes approximately one microsecond per degree of polynomial.

SYNCHRONOUS SYSTEM BUS

The SBI (Synchronous Backplace Interconnect) is the system's internal backplane and bus which conveys addresses, data, and control information between the processor and memory, and between memory and peripheral controllers. The SBI has a cycle time of 200 nanoseconds and can transfer 32 bits each cycle. Data transfers use two consecutive cycles to transfer 64 bits at a time. The maximum SBI transfer rate is 13.3 million bytes per second. The SBI provides an unusual degree of throughput and reliability because it uses:

- time-division multiplexing
- distributed priority arbitration
- parity and protocol checking on every transfer
- transaction history recording

The protocol, or sequence in which operations occur on the SBI, is time-division multiplexed to increase the effective bus bandwidth. Time-division multiplexing means that the transactions consituting one transfer operation are interleaved with the transactions constituting another transfer operation. Thus, several operations can be in progress over the same period of time. For example, the CPU can ask a memory controller to read some data; the same memory controller might then transfer previously requested data to an I/O device before transferring the requested data to the CPU.

In some systems, the processor bus can be tied up for each transfer because a requester acquires the bus to send an address and then keeps the bus while it waits for the requested data. In VAX-11/780, the bus is not held inactive during the data access time because bus ownership is relinquished after every cycle. A requester acquires the bus to send an address, relinquishes the bus, and then the responder acquires the bus to send the data. In the interim, any number of other transactions can be initiated or completed. This and the fact that transactions are buffered make it possible for the bus to operate at its full bandwidth, because a bus transaction can take place every 200 nanoseconds.

Arbitration on the SBI is distributed, which ensures that no unit is critical to bus operation. Every unit on the SBI has its own arbitration line. Arbitration lines are ordered by priority and every unit monitors all the arbitration lines each cycle to determine if it will get the next cycle. Unlike some bus systems, any unit on the SBI (except the CPU clock) can fail without causing a failure of the entire bus.

To ensure the integrity of the signals transmitted, the SBI includes several error checking and diagnostic mechanisms, such as:

- parity checking on data, addresses, and commands
- protocol checking in each interface
- a history silo of the last 16 SBI cycles

MAIN MEMORY AND CACHE SYSTEMS

The processor includes both main memory systems and cache memory systems. Transactions between main memory and the processor take place over the SBI. The cache memory systems are internal to the processor.

Main Memory

Main memory consists of arrays of MOS RAM integrated circuits with a cycle time of 600 nanoseconds. A memory controller can access a maximum of 4,194,304 bytes (4M bytes). Two memory controllers can be connected to the SBI, yielding a maximum of 8M bytes of physical memory that can be available on the system. (The maximum total physical address space is 2^{29} or approximately 512 million bytes.) The minimum required memory is 256K bytes, which can be built up to the maximum in increments of 256K bytes.

A memory controller will buffer one command while it processes another to increase system throughput. Main memory can also be interleaved (where two memory controllers are each addressing the same amount of memory) to increase the available memory bandwidth. The memory system employs error checking and correction (ECC) that corrects all single bit errors and detects all double bit errors.

When the system is powered down, an AC standby current is normally used to retain the contents of memory. In case of temporary AC power interruption, an optional backup battery is also available to provide 10 minutes of power for up to 4M bytes of memory so that the contents of main memory are not destroyed. Two backup batteries provide power for up to 8M bytes of memory.

Whenever possible, data are fetched from main memory 64 bits at a time (two SBI cycles) and cached in the processor's internal memory systems. The internal memory systems include a main memory cache, an address translation buffer, and an instruction lookahead buffer.

Memory Cache

The memory cache is the primary cache system for all data coming from memory, including addresses, address translations; and instructions. The memory cache is an 8K byte, two-way set associative, write-through cache.

Write-through provides reliability because the contents of main memory are updated immediately after the processor performs a write. Where most write-through cache systems tie up the processor while main memory is updated, however, this processor buffers its commands to avoid waiting while main memory is updated from the cache. Therefore, while providing the reliability of a write-through cache, this system also provides much the same performance as a write-back cache.

The memory cache also reduces the average time the processor waits to receive main memory data by reading eight bytes at a time from main memory, and transferring four bytes to the CPU data paths. Since the remaining four bytes are already available, the memory cache also provides pre-fetching. The cache memory system carries byte parity for both data and addresses for increased integrity.

Address Translation Buffer

The address translation buffer is a cache of likely-to-be-used physical address translations. It significantly reduces the amount of time spent by the CPU on the repetitive task of dynamic address translation. The cache contains 128 virtual-to-physical page address translations which are divided into equal sections: 64 system space page translations and 64 process space page translations. Each of these sections is two-way associative. There is byte parity on each entry for increased integrity.

Instruction Buffer

The 8-byte instruction buffer improves CPU performance by prefetching data in the instruction stream. The control logic continuously fetches data from memory to keep the 8-byte buffer full. It effectively eliminates the time spent by the CPU waiting for two memory cycles where bytes of the instruction stream cross 32-bit longword boundaries. In addition, the instruction buffer processes operand specifiers in advance of execution and subsequently routes them to the CPU.

PERIPHERAL CONTROLLER INTERFACES

Peripherals can be connected to the processor's SBI in either of two ways: through the MASSBUS, which conveys the signals to and from high-speed disk or magnetic tape devices, or through the UNIBUS, which conveys the signals to and from a variety of I/O devices, including line printers, disks, card readers, terminals, and interprocessor communication links.

MASSBUS Interface

The processor interface for a MASSBUS peripheral is the **MASSBUS adaptor**. The MASSBUS adaptor performs control, arbitration, and buffering functions. Up to four MASSBUS adaptors can be placed on the SBI.

Each MASSBUS adaptor includes its own address translation map that permits scatter/gather disk transfers. In scatter/gather transfers, physically contiguous disk blocks can be read into or written from discontiguous blocks of memory. The translation map contains the addresses of the pages, which may be scattered throughout memory, from or to which the contiguous disk transfer takes place.

Each MASSBUS adaptor includes a 32-byte silo data buffer. Data are assembled in 64-bit quadwords (plus parity) to make efficient use of the SBI bandwidth. On transfers from memory to a MASSBUS peripheral, the MASSBUS adaptor anticipates upcoming MASSBUS data transfers by fetching the next 64 bits from memory before all of the previous data are transferred to the peripheral.

On-line diagnostics and loop-back enable adaptor fault isolation for any function circuits without requiring the use of a drive on the MASSBUS.

UNIBUS Interface

All devices other than the high-speed disk drives and magnetic tape transports are connected to the UNIBUS, an asynchronous bi-directional bus. These include all DIGITAL- and user-developed real-time peripherals. The UNIBUS is connected to the SBI through the **UNIBUS adaptor**. The UNIBUS adaptor does priority arbitration among devices on the UNIBUS.

The UNIBUS adaptor provides access from the processor to the UNIBUS peripheral device registers and to UNIBUS memory by translating UNIBUS addresses, data, and interrupt requests to their SBI equivalents, and vice versa. The UNIBUS adaptor address translation map translates an 18-bit UNIBUS address to a 30-bit SBI address. The map provides direct access to system memory for non-processor request UNIBUS peripheral devices and permits scatter/gather disk transfers.

The UNIBUS adaptor enables the processor to read and/or write the peripheral controller status registers. In the case of processor interrupt request devices, this constitutes the transfer.

To make the most efficient use of the SBI bandwidth, the UNIBUS adaptor provides buffered direct memory access data paths for up to 15 non-processor request (NPR) devices. Each of these channels has a 64-bit buffer (plus byte parity) for holding four 16-bit transfers to and from UNIBUS devices. The result is that only one SBI transfer (64 bits) is required for every four UNIBUS transfers. The maximum aggregate transfer rate through the buffered data paths is 1.5 million bytes per second. On SBI-to-UNIBUS transfers, the UNIBUS adaptor anticipates upcoming UNIBUS requests by pre-fetching the next 64-bit quadword from memory as the last 16-bit word is transferred from the buffer to the UNIBUS. By the time the UNIBUS device requests the next word, the UNIBUS adaptor has it ready to transfer.

Any number of unbuffered direct memory access transfers are handled by one Direct Data Path. Every 8- or 16-bit transfer requires one 32-bit transfer on the SBI. The maximum transfer rate through the Direct Data Path is 500 thousand bytes per second.

The UNIBUS adaptor permits concurrent program interrupt, unbuffered, and buffered data transfers. The aggregate throughput rate of the Direct Data Path, plus the 15 buffered data paths, is 1.5 million bytes per second.

5 The Peripherals

The VAX-11/780 system supports high performance mass storage devices for on-line data retrieval, unit record equipment for data processing, terminals and line interfaces for the interactive user, Direct Memory Access interfaces for real-time users and a line interface for interprocessor communications.

The mass storage systems provide large capacity and rapid throughput. As many as four MASSBUS adaptors, each of which can support up to eight disk drives or magnetic tape controllers, can be connected to the system. In addition, up to eight medium-capacity disk drives can be connected to the system's UNIBUS. The software overlaps seeks on all multiple-drive disk configurations, performs multiple-block I/O transfers, and allows the user to control buffering and blocking.

Card readers and line printers can be spooled input and output devices managed by operator-controlled queues. LP11 and LA11 series line printers provide a range of high-speed and low-cost printer models to choose from. Up to four LP11 printers and up to 16 LA11 printers can be available on the system.

The system supports full-duplex handling for both hard copy and video terminals. The LA120 is a hard-copy terminal offerring exceptional throughput and advanced print features; the VT100 video terminal offers a variety of controllable character and screen attributes including (24 lines by 80 columns) or (14 lines by 132 columns) screen sizes, smooth scrolling, and split screen. The system can support up to 96 terminal lines.

The DMC11 serial synchronous communications line provides high-performance point-to-point interprocessor connection using the DIGITAL Data Communications Message Protocol (DDCMP). The DMC11 ensures reliable data transmission and relieves the host processor of the details of protocol operation.

For real-time applications, VAX supports the LPA11-K and DR11-B direct memory access (DMA) interfaces. These devices reduce CPU involvement in I/O operations and speed the transfer of data between external devices and computer memory. The LPA11-K is an intelligent (two-microprocessor) controller which provides high speed data sampling, operates in both dedicated and multirequest mode, and supports a number of peripheral devices. The DR11-B is a general purpose interface which performs high speed block data transfers between the VAX UNIBUS and user peripheral devices.

All equipment is integrated with the software system, and is supported by both on-line error logging and diagnostics. Each component includes extensive error checking and correction features. The software provides power failure and error recovery algorithms.

COMPONENTS

VAX-11/780 supports four types of peripheral systems:

- mass storage peripherals such as disk and magnetic tape
- unit record peripherals such as line printers and card readers
- terminals and terminal line interfaces
- interprocessor communications links

All peripheral device control/status registers (CSRs) are assigned addresses in physical I/O space. No special processor instructions are needed for I/O control. In addition, all device interrupt lines are associated with locations that identify each device's interrupt service routine. When the processor is interrupted on function request completion, it immediately starts executing the appropriate interrupt service routine. There is no need to poll devices to determine which device needs service.

Devices use either one of two types of data transfer techniques: direct memory access or programmed interrupt request. The mass storage disk and magnetic tape devices and the interprocessor communications link are capable of direct memory access (DMA) data transfers. The DMA devices are also called **non-processor request** (NPR) devices because they can transfer large blocks of data to or from memory without processor intervention until the entire block is transferred.

The unit record peripherals and terminal interfaces are called **programmed interrupt request** devices. These devices transfer one or two bytes at a time to or from assigned locations in physical address space. Software then transfers the data to or from a buffer in physical memory.

MASS STORAGE PERIPHERALS

The mass storage peripherals include three types of moving head disk drive and one type of magnetic tape transport:

- the high speed, large capacity RP06 disk drive
- the high speed, medium capacity RM03 disk drive
- the small capacity RK06 and RK07 disk drives
- the TE16 and TU45 magnetic tape transports

The RP06 and RM03 disk and the TE16 and TU45 magnetic tape controllers are MASSBUS peripheral devices. It is possible to connect as many as four MASSBUS adaptors to the central processor. Each MASSBUS can support up to eight logical devices. Each disk drive and magnetic tape formatter counts as a logical device. Hence, up to eight disk drives and tape formatters in any combination may be attached.

To support the performance and reliability features of the system's disk and magnetic tape devices, the operating system's disk and magnetic tape device drivers provide:

- overlapped seeks for increased throughput on controllers with multiple disk drives
- overlapped magtape operations (write on one transport while another rewinds, for example)
- multiple block non-contiguous I/O transfers for file-structured devices
- read and write checks on a per-request, per-file, and/or volume basis
- extensive error recovery algorithms (e.g., ECC and offset recovery for disk, NRZI error correction for magnetic tape)
- logging of all device errors
- dynamic bad block support for file-structured disk devices
- volume mount verification after a change in drive status (off/on-line, powerfail)
- powerfail recovery for on-line drives, including repositioning of magnetic tape transports

For applications requiring special data reliability checks, programmers can implement their own error recovery procedures without having to write their own device driver routines. The operating system driver's normal error recovery retry and error logging operations can be inhibited. If any error occurs when the recovery functions are inhibited, the driver immediately terminates the I/O operation and returns a failure status. User software can then perform its own recovery or logging procedures, since all the hardware diagnostic operations are available to jobs granted the diagnostic privilege by the system manager.

Disks

The disk subsystems all provide high performance and reliability. They feature accurate servo positioning, error correction, and offset positioning recovery. Table 5-1 summarizes the capacities and speeds of the disk devices.

Table 5-1
Disk Devices

DISK	RK06	RK07	RM03	RP06
Pack capacity:	14 Mb	28 Mb	67 Mb	176 Mb
Peak transfer rate (/sec):	538 Kb	538 Kb	1200 Kb	806 Kb
Ave. seek time:	38ms	36.5ms	30ms	28ms
Ave. rotational latency:	12.5ms	12.5ms	8.3ms	8.3ms

All disk drives use top-loading removable media. The RM03 and RP06 disk drives can be mixed on the same MASSBUS controller. Each controller can handle up to eight disk drives.

The UNIBUS accepts both RK06 and RK07 disk drives. As illustrated in Table 5-1, these disk drives show similar access times and transfer rates, but the RK07 has twice the capacity. Up to eight RK07 and RK06 drives can be mixed in any combination on the same controller. Either drive can be used as the systems device. In small system configurations where the RK06 or RK07 is used as the systems device, two drives are required in the configuration.

To decrease the effective access time and increase throughput, the operating system's disk device drivers provide overlapped seeks for all disk units on a controller. All I/O transfers, including write checks, are preceded by a seek, except when the seek is explicitly inhibited by diagnostic software. On MASSBUS devices, seeks to any unit can be initiated at any time and do not require controller intervention. During seeks, the controller is free to perform a transfer on any unit other than the one on which the seek is active. If a data transfer was in progress at the time of completion, the driver processes the attention interrupts caused by seek completion when the controller is free.

The device unit notifies the driver when it detects a read error that can be recovered using its error correction code (ECC). It provides the position and pattern of any error burst of up to 11 bits within the data field. The driver applies the error correction to the data in memory. The transfer continues as if the error had not occurred.

In addition to overlapping seeks with data transfers, the driver also overlaps offset error recovery with normal controller operation. Offset recovery enables the driver to reposition the head on the track to pick up a stronger signal on a sector during a read operation. Provided that retry is not inhibited, the driver performs offset recovery automatically when a read error occurs that can not be corrected using the hardware ECC.

The driver logs all errors, including those from which it successfully recovers. The driver also supplies dynamic bad block handling for virtual I/O (Files-11 file-structured) operations. When a bad block is detected, the information is stored in the file header. The bad block is recorded in the bad block file when the file is deleted.

In addition to the driver's dynamic bad block handling, the system includes an on-line static bad block utility and on-line diagnostics for verifying drive level functions.

Magnetic Tape

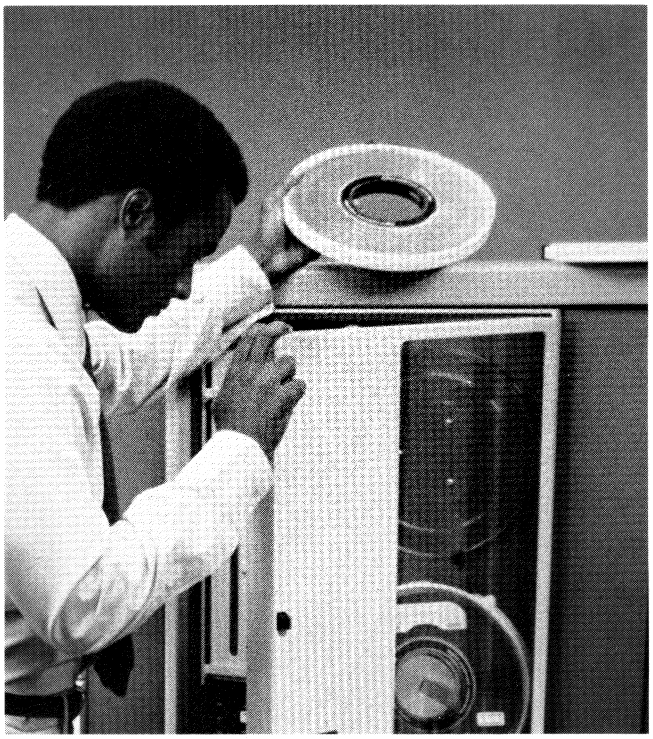
The TE16 and TU45 are high performance tape storage systems, which share the following characteristics:

- Program-selectable 1600 or 800 bpi, 9-track data storage
- Industry compatible data formats
- Reading in reverse (as well as forward)
- Parity, longitudinal, and cyclic redundancy checking
- NRZI error correction

The TU45 and TE16 are identical in capacity: both allow up to 40 million bytes per tape reel and both allow up to 8 tape drives per formatter. However, the TU45 offers substantially higher speed and throughput. Read/write speeds for the TU45 and TE16 are 75 and 45 inches per second respectively; their data transfer speeds are 120K and 72K bytes per second.

The operating system's magnetic tape device driver supports the read reverse operation, which enables a program to request a sequential read of the block preceding the block at which the tape is positioned. Writing occurs only while the tape is moving forward.

The operating system's file system can read and write file-structured magnetic tape volumes using the ANS Level 3 tape labels and format. The system also supports multivolume files, program-controlled blocking factors, and unlabeled magnetic tapes.



UNIT RECORD PERIPHERALS

The operating system normally treats line printers and card readers as spooled sharable devices managed by multiple operator-controlled queues. The devices can also be allocated to individual programs.

The operating system's line printer handling includes line and page counting for job accounting. The user can specify carriage control as: one line per record, FORTRAN conventions, contained within the record itself, or general pre- and post-spacing (within the limits of the hardware capabilities).

The operating system's card reader driver interprets the encoded punched information using the American National Standard 8-bit card code. The driver uses a special punch outside the data representation to indicate end-of-file.

LP11 Line Printers

Up to four LP11 series line printers can be connected to the VAX-11/780 system. The LP11 series printers are impact-type, rotating drum, serial interface line printers. They feature full line buffering, a static eliminator, and a self-test capability.

All models are 132-column printers that can accept paper 4 to 16-3/4 inches wide with up to 6-part forms. They print

10 characters per inch horizontally, and 6 or 8 lines per inch vertically (switch selectable). They include a vernier adjustment for horizontal and vertical paper position. All models are available with either upper (64) or upper/lower (95) character sets (including numbers and symbols). Most models have optional scientific or EDP character sets.

The low-cost models print one line every two revolutions (300 lines per minute with the 64-character set, 230 lines per minute with the 95-character set), or one line every revolution (600 lines per minute with the 64-character set, 460 lines per minute with the 95-character set). A higher-speed version that includes a noise-reduction cabinet, ribbon guide, and a high-speed paper puller offers 900 line-per-minute printing with the 64-character set, or 660 lines per minute with the 95-character set.

For systems requiring even greater printer throughput, LP11 models are available that print up to 1200 lines per minute with the 64-character set or 800 lines per minute with the 95-character set.

LA11 Line Printer

Up to 16 LA11 line printers can be connected to the system. The LA11 is an extremely low-cost, highly reliable parallel interface printer. The LA11 prints at speeds up to 180 characters per second. The print set consists of the ASCII characters, including 95 upper and lower case letters, numbers, and symbols. Characters are printed using a 7 x 7 matrix with horizontal spacing of 10 characters per inch and vertical spacing of six lines per inch.

Adjustable pin-feed tractors allow for a variable-form width of 3 to 14-7/8 inches (up to 132 columns). A forms length switch sets the top-of-form to any of 11 common lengths, with fine adjustment for accurate forms placement. The printer can accommodate multi-part forms (with or without carbons) of up to six parts.

CR11 Card Reader

Up to two CR11 card readers can be connected to the system as programmed interrupt request devices. The CR11 reads up to 285 80-column punched cards per minute. The card reader has a high tolerance for cards that have been nicked, warped, bent, or subjected to high humidity. The card reader uses a short card path, with only one card in the track at a time. It uses a vacuum pick mechanism and keeps cards from sticking together by blowing a stream of air through the bottom half-inch of cards in the input hopper. The input hopper holds up to 400 cards, and cards can be loaded and unloaded while the reader is operating.

TERMINALS AND INTERFACES

Up to 96 interactive terminals can be connected to the VAX-11/780 system. The operating system's terminal driver provides full duplex handling for both hard copy and video terminals.

Programs can control terminal operations through the terminal driver. The terminal driver supports many special operating modes for terminal lines. A program can enable or disable the following modes by calling a system service:

- SLAVE All unsolicited data are discarded. This mode is used to establish application-controlled terminals.
- NO ECHO Data entered on the terminal keyboard are not printed or displayed on the terminal. This mode is used, for example, to read passwords typed on the terminal.
- PASS ALL All data entered on the terminal are transmitted to the program as 8-bit binary information without any interpretation, except where a line terminator or terminators are specified. This mode enables programs to perform their own interpretation of control characters instead of using the VAX/VMS interpretation.
- ESCAPE Escape sequences entered on the terminal are recognized as read terminators, validated, and passed to a program for interpretation.
- TERMINAL/HOST SYNCHRONIZATION Data sent to the terminal are controlled by terminal-generated XOFF and XON. These functions are generated by typing CTRL/S and CTRL/Q on command terminals and are interpreted as requests to stop and resume output to the terminal.
- HOST/TERMINAL SYNCHRONIZATION All read operations are explicitly solicited with XON and terminated with XOFF. XON and XOFF are also used to keep the type-ahead buffer from filling.

Input from a command terminal is always independent of concurrent output. This capability is called *type-ahead*. Data typed at the terminal are retained in a type-ahead buffer until a program issues a read request. At that time the data are transferred to the program buffer and echoed on the terminal (provided that echoing is not disabled). If a read is already in progress, the echo and data transfer are immediate. Deferring the echo until a read operation is active allows the program to specify the mode of the terminal, such as No Echo or Convert Lower Case to Upper, to modify the read operation.

A line entered on a command terminal is terminated by any of several special characters, for example, the RETURN key. A program reading from a terminal can optionally specify a particular line terminator or class of line terminators for read requests (including read PASS ALL requests).

Terminal characteristics are initially established during system generation. Users operating command terminals can modify the characteristics of the particular terminal being used. For example, the user can set the baud rate (transmission speed) or change the terminal line width.

LA120 Hard Copy Terminal

The LA120 is a hard copy terminal which offers exceptional throughput and a number of advanced keyboard-selectable formatting and communication features. It uses a contoured typewriter-styled keyboard and includes an additional numeric keypad and a prompting LED display for infrequently used features.

The LA120 achieves high throughput owing to several features:

- 180 character per second print speed
- 14 data transmission speeds ranging up to 9600 baud
- 1K character buffer to equalize differences between transmission speeds and print speeds

- smart and bidirectional printing so that printhead always takes shortest path to next print position
- high speed horizontal and vertical skipping over white space

In addition to its throughput, the LA120 is distinguished by its printing features. The terminal offers 8 font sizes, ranging from expanded (5 characters per inch) to compressed font (16.5 characters per inch). Hence a user could, for example, select a font size of 16.5 cpi and print 132 columns onto an 8 1/2-inch-wide sheet. Other print features include 6 line spacings ranging from 2 to 12 lines per inch, user-selectable form lengths up to 14 inches, left/right and top/bottom margins and horizontal and vertical tabs.

The LA120 is designed for easy use. Terminal characteristics are selected via clearly labeled keys and simple mnemonic commands. Once the selections have been made, the operator can check his settings by depressing the STATUS key. The terminal will then print a listing of the selected settings.

LA36 Hard Copy Terminal

The LA36 is an exceptionally reliable hard copy terminal. It is a lower-priced device than the LA120 with lower throughput (30 cps vs. 165 cps) and fewer print features.

The LA36 uses a typewriter-like keyboard which produces 128 ASCII characters, consisting of 95 upper- and lower-case printing characters and 33 control characters, and is available with optional special character sets, including various foreign language character sets.

Characters are printed using a 7 x 7 matrix with horizontal spacing of 10 characters per inch and vertical spacing of six lines per inch. To ensure clear visibility of the printed line, the print head automatically retracts out of the way when not in operation. Adjustable pin-feed tractors allow for a variable-form width from 3 to 14 7/8 inches (up to 132 columns). The print mechanism will accommodate multi-part forms (with or without carbons) of up to six parts.

The LA36 operates at speeds of 110, 150, or 300 baud (10, 15, or 30 characters per second). Printable characters are stored in a buffer during the carriage return operation. While more than one character is in the buffer, the printer mechanism operates at an effective speed of 60 characters per second.

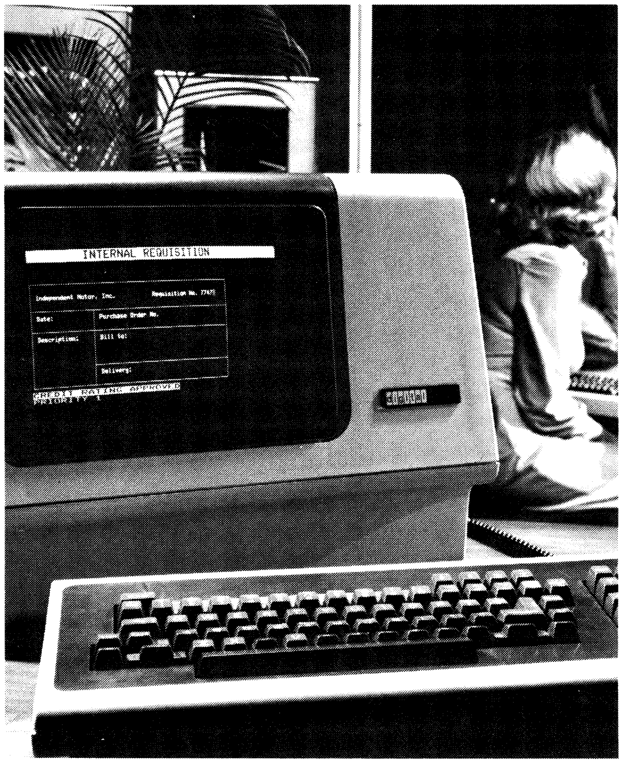
VT100 Video Terminal

The VT100 Video Terminal is an upper- and lower-case ASCII terminal which offers a variety of controllable character and screen attributes. The VT100 features a typewriter-like detachable keyboard which includes a standard numeric/function keypad for data entry applications. Also featured are seven LEDs, four of which are program-controlled, used as operator information and diagnostic aids.

The VT100 offers a number of advanced features. The most important of these are:

- Ability to select either of two screen sizes: 24 lines by 80 columns or 14 lines by 132 columns
- Ability to select either double-width single-height characters or double-width double-height characters on a line by line basis
- Smooth scrolling and split screen capability.
- Ability to set baud rates, tabs, and Answer Back messages from the keyboard and to store these in RAM (Random Access Memory)
- Special line drawing graphic characters providing the ability to display simple graphics for business or laboratory applications
- Ability to select black-on-white characters or white-on-black characters on a full screen basis.

In addition, several options further extend the capabilities of the VT100. These include the advanced video option, which adds selectable blinking, underline, and dual intensity characters to the existing reverse video attribute; the provision of space, power, and interconnects for the future addition of a terminal processor; and additional RAM allowing 24 lines of 132 characters.



DZ11 Terminal Line Interface

The DZ11 is a serial line multiplexer whose character formats and operating speeds are programmable on a per-line basis. A DZ11 connects the UNIBUS with up to a maximum of 8 or 16 asynchronous serial lines. Each line can run at any one of 15 speeds.

Local operation with EIA terminals is possible at speeds up to 9600 baud. Remote dial-up terminals can operate full duplex at speeds up to 300 baud using Bell 103 or 113 modems, or up to 1200 baud using the Bell 212 modem.

The DZ11 optionally generates parity on output and checks parity on input. Incoming characters are buffered using a 64-character silo buffer. Outgoing characters are processed on a programmed interrupt request basis.

REAL-TIME I/O DEVICES

To enhance real-time performance, VAX supports the DR11-B and LPA11-K direct access memory (DMA) interfaces. These devices allow data to be transferred from a peripheral device to memory and vice-versa without the intervention of computer programs except at the initialization and completion of transfers. The result is that CPU involvement in I/O operations is greatly reduced. Further, since these devices are capable of "driving" large blocks of information at rapid speeds, their usage can greatly increase I/O bandwidth, i.e., the capacity of the system to sustain a total data transfer load. I/O bandwidth is an important performance measure in real-time applications, since such applications require complete data transfers between external devices and computer memory.

The LPA11-K

The LPA11-K is an intelligent (two-microprocessor) direct memory access (DMA) controller that buffers real-time data and transfers it to VAX memory in efficient blocks (rather than a word at a time). Since the LPA11-K has automatic buffer switching capability, transfers occur continuously with minimum interruptions. Via a system call, the programmer can instruct the LPA11-K to take samples from a data source at specified time intervals. Sampling is handled by the microprocessors, without the intervention of the CPU. Under VMS, the LPA11-K can be accessed via FORTRAN IV-PLUS, BASIC-PLUS-2, BLISS-32, and MACRO.

The LPA11-K operates in two distinct modes: dedicated mode and multirequest mode:

In multirequest mode, up to eight users can be active concurrently. Each user's sampling rate is a user-specified multiple of the common real-time clock rate; thus independent rates can be maintained for each user. Each request specifies the device so that A/D, D/A or digital I/O can be synchronously sampled; the transition of a bit in a digital word can synchronize the sampling with a user event. In multirequest mode, the throughput of data is determined by the number and types of requests. The aggregate throughput rate for all users is typically 15,000 samples per second.

In dedicated mode, one user can sample from analog-to-digital converters, or output to a digital-to-analog converter. Two analog-to-digital converters can be controlled simultaneously. Sampling is initiated by an overflow of the real-time clock, or by an external signal. Two sampling algorithms are implemented. One, at each overflow, samples both analog-to-digital converters in parallel, allowing two channels to be sampled simultaneously. The other algorithm samples the two converters on an interleaved basis, beginning with the first whose sampling begins on alternate clock overflows.

The LPA11-K supports the following I/O devices on VAX.

- DA11K (D/A converter)
- AD11K (eight-channel A/D converter)
- AM11K (multiplexer board)
- DR11K (general device interface)
- KW11K (real-time clock)

The DR11-B

The DR11-B is a general purpose, direct memory access (DMA) digital interface to the UNIBUS. The DR11-B, rather than using program controlled data transfers, operates directly to or from memory, moving data between the UNIBUS and the user device. The DR11-B, like the LPA11-K, is a block transfer device. However, it is less expensive than the LPA11-K, does not include a microprocessor, and can only handle a single task for a single programmer.

The DR11-B interface consists of four registers: command and status, word count, bus address, and data. Operation is initialized under program control by loading word count with the 2's complement of the number of transfers, specifying the initial memory or bus address where the block transfer is to begin and by loading the command/status register with function bits. The user device recognizes these function bits and responds by setting up the control inputs. If the user device requests data from memory of a UNIBUS device, the DR11-B performs a UNIBUS Data In transfer (DATI) and loads its data register with the information held at the referenced bus address. The outputs of this register are available to the user device; this output data is buffered. If the user device requests data to be written into memory, the DR11-B performs a UNIBUS Data Out transfer (DATO), moving data from the user device to the referenced bus address; this input data is not buffered. Transfers normally continue at a user-defined rate until the specified number of words is transferred. The DR11-B has the capability of transferring data at a rate of 500,000 bytes per second, but actual transfer rates depend upon the particular configuration.

INTERPROCESSOR COMMUNICATIONS LINK

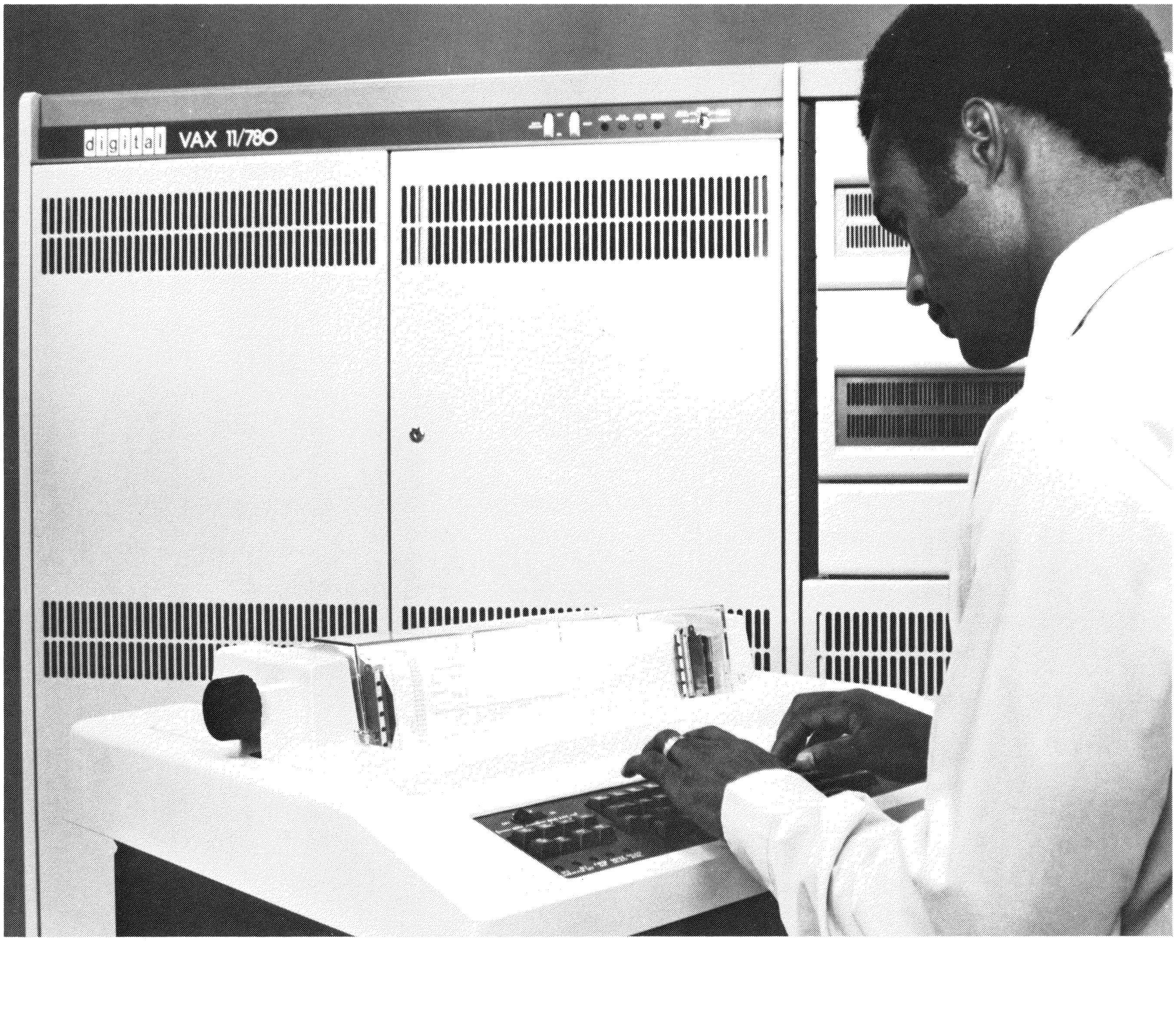
The DMC11 communications link is designed for high-performance point-to-point interprocessor connection based on the DIGITAL Data Communications Message Protocol (DDCMP). The DMC11 provides local or remote interconnection of two computers over a serial synchronous link. Both computers can include the DMC11 and DECnet software, or both computers can include the DMC11 and implement their own communications software. For remote operations, a DMC11 can also communicate with a different type of synchronous interface provided that the remote system has implemented the DDCMP protocol.

By implementing the DDCMP protocol in its high-speed microprocessor, the DMC11 ensures reliable data transmission and relieves the host processor of the details of protocol operation (including character and message synchronization, header and message formatting, error checking, and retransmission control). The DDCMP protocol detects errors on the channel interconnecting the system using a 16-bit Cyclic Redundancy Check (CRC-16). Errors are corrected by automatic retransmission. Sequence numbers in message headers ensure that messages are delivered in proper order with no omissions or duplications.

The DMC11 supports full- or half-duplex operation. Full-duplex operation offers the highest throughput and is used when the communications facilities permit two-way operation. The DDCMP protocol permits continuous simultaneous transmission of data messages in both directions when buffers are available and there are no errors on the channels.

Where both computers are located in the same facility, the DMC11 permits transmission at speeds of up to 1,000,000 bits per second over coaxial cable up to 6,000 feet long, or speeds of up to 56,000 bits per second over coaxial cable up to 18,000 feet long. The necessary modems for local interconnection are built in. Where the computers are located remotely and connected using common carrier facilities, the DMC11 permits transmission of up to 19,200 bits per second using an EIA interface. A DMC11 can interface to synchronous modems such as the Bell models 208 and 209, or other synchronous modems conforming to the RS232-C standard.

6 The Operating System



VAX/VMS is the general purpose operating system for the VAX-11/780. It provides a reliable, high-performance environment for the concurrent execution of multiuser timesharing, batch, and real-time applications. VAX/VMS provides:

- virtual memory management for the execution of large programs
- event-driven priority scheduling
- shared memory, file, and interprocess communication data protection based on ownership and application groups
- programmed system services for process and subprocess control and interprocess communication

VAX/VMS uses the VAX-11/780 memory management features to provide swapping, paging, and protection and sharing of both code and data. Memory is allocated dynamically. Applications can control the amount of physical memory allocated to executing processes, the protection of pages, and swapping. These controls can be added after the application is implemented.

VAX/VMS schedules CPU time and memory residency on a preemptive priority basis. Thus, real-time processes do not have to compete with lower priority processes for scheduling services. Scheduling rotates among processes of the same priority.

VAX/VMS allows real-time applications to control their virtual memory paging and execution priority. Real-time applications can eliminate services not needed to reduce system overhead. Processes granted the privilege to execute at real-time scheduling levels, however, do not necessarily have the privilege to access protected memory and/or data structures.

VAX/VMS includes system services to control processes and process execution, control real-time response, control scheduling, and obtain information. Process control services allow the creation of subprocesses as well as independent detached processes. Processes can communicate and synchronize using mailboxes, shared areas of memory, or shared files. A group of processes can also communicate and synchronize using multiple common event flag clusters.

Applications designers can use the VAX/VMS protection and privilege mechanisms to implement system security and privacy. VAX/VMS provides memory access protection both between and within processes. Each process has its own independent virtual address space which can be mapped to private pages or shared pages. A process can not access any other process' private pages. VAX/VMS uses the four processor access modes to read and/or write protect individual pages within a process. Protection of shared pages of memory, files, and interprocess communication facilities such as mailboxes and event flags, is based on User Identification Codes individually assigned to accessors and data.

VAX-11/780 is built for executing high-performance applications where:

- Event-driven interprocess communication and procedure and data sharing are important. Order entry and teller transaction systems often consist of many cooperating processes that synchronize record creation and modification.
- Priorities of resource allocation can be set for currently executing jobs. Both real-time processess and resource-sharing processes can execute in the same environment, as in a communications network. High-speed links can be serviced on demand, while interactive terminal users and batch jobs share processor time and peripherals.
- Large programs can be developed to execute in a physical memory smaller than the program's total memory requirements. Engineering computation programs such as simulators often build data arrays which require a large address space to describe the arrays.

The VAX operating system provides the run-time services for executing high-performance application systems. Operations managers and systems programmers have considerable flexibility in designing and controlling data and program flow.

Applications can be divided into several independent subsystems whose data and code are protected from one another, and yet have general communication and data sharing facilities. Jobs can communicate using general, group, or local communication facilities.

Applications which require an immediate response to some external event can be scheduled as real-time processes. When a real-time process is ready to execute, it executes until it becomes blocked or another higher priority real-time process needs the resources of the processor. Normal jobs can be scheduled using a modified preemptive algorithm that ensures that they receive processor and peripheral resources at regular intervals commensurate with their processing needs.

If insufficient memory is available for keeping concurrently executing jobs resident, the operating system will swap jobs in and out of memory to allocate each its share of processor time. Real-time processes can be locked in memory to ensure that they can be started up rapidly when they need to execute.

The operating system provides a dynamic virtual memory programming environment. Large programs can be executed in a portion of physical memory that is considerably smaller than the program's memory requirements, without requiring the programmer to define overlays. The operating system optimizes its virtual memory system for program locality and provides tools that support optimization. It makes program performance predictable and controllable by restricting paging to the process running a large program, and by bringing in large amounts of a program at one time.

The operating system provides sophisticated peripheral device management for sharing, protection, and throughput. Devices can be shared among all jobs or reserved for exclusive use by particular jobs. Input and output for low-speed devices is spooled to high-speed devices to increase throughput. Files on mass storage devices can be protected from unauthorized access on an individual, group, or volume basis.

Furthermore, the I/O request processing system is optimized for throughput and interrupt response. The operating system provides the programmer with several data accessing methods, from logical record accessing for easy, device-independent programming, to direct I/O accessing for extremely rapid data processing. Files can be stored in any of several ways to optimize subsequent processing.

VAX provides the programming tools, scheduling services, and protection mechanisms for multiuser program development. Programmers can write, execute, and debug programs on the system interactively, and also create batch command files that perform repetitive program development operations without requiring their attention.

Although it provides a multiuser program development environment, VAX is unlike traditional program development timesharing systems. VAX is an application execution system that optimizes total system throughput and response to high-priority activities. As in a timesharing system, interactive jobs can be given equal opportunities for resource acquisition. In addition, the system can be executing real-time applications while program development jobs run, since higher priority activities always have the ability to preempt lower priority activities.

COMPONENTS AND SERVICES

The operating system is the collection of software that organizes the processor and peripherals into a high-performance application execution system. The operating system's basic components include:

- processes that control initial resource allocation, communicate with the system operator, and log errors
- the command interpreters
- user-programmed process control services
- exception dispatcher
- memory management routines for program image activation and paging
- scheduling routines and swapper
- file and record management services
- interrupt and I/O processing routines
- compatibility mode executive routines

The operating system's jobs run as independent activities on the system. They include the Job Controller, which initiates and terminates user processes and manages spooling, the Operator Communications Manager, which handles messages queued to the system operators, and the Error Logger, which collects all hardware and software errors detected by the processor and operating system.

A command interpreter executes as a service for interactive and batch jobs. It enables the general user to request the basic functions that the operating system provides, such as program development, file management, and system information services.

Both hardware-detected and software-detected exception conditions are tracked through the exception dispatcher. The exception dispatcher passes control to user-programmed condition handlers or, in the case of system-wide exception conditions, to operating system condition handlers.

The operating system's memory management routines include the image activator, which controls the mapping of virtual memory to system and user jobs, and the pager, which moves portions of a process in and out of memory as required. They respond to a program's dynamic memory requirements, and enable programs to control their allocated memory, share data and code, and protect themselves from one another.

The scheduler controls the allocation of processor time to system and user jobs. The scheduler always ensures that the highest priority, ready-to-execute real-time process receives control of the processor until it relinquishes it. When no real-time processes are ready to execute, the scheduler dynamically allocates processor time to all other jobs according to their priorities and resource requirements. The swapper works in conjunction with the scheduler to move entire jobs in and out of memory when memory requirements exceed memory resources. The swapper ensures that the jobs most likely to execute are kept in memory.

The operating system's I/O processing software includes interrupt service routines, device-dependent I/O drivers, device-independent control routines, and user-programmed record processing services. The I/O system ensures rapid interrupt response and processing throughput, and provides programming interfaces for both special purpose and general purpose I/O processing.

The next few sections discuss some of the concepts basic to understanding the operating system's functions and services. They are followed by descriptions of the services available to individual and cooperating processes, and descriptions of memory management and scheduling for the systems programmer.

PROCESSING CONCEPTS

To support high-performance multiprogramming application systems, the operating system provides the applications programmer with the tools to implement:

- shared programs
- shared files and data
- interprocess communication and control

To enable the programmer to write shared programs easily, the operating system treats a program independently of the context in which a program executes. The context defines the privileges assigned by the system manager to a particular user. Users with different privileges can share programs, and the operating system will enforce protection independent of the program.

The operating system controls privilege and accounts for resource allocation by job. A job can be performing processing operations under the direction of one user at a terminal, or it can be performing processing operations for several users at multiple terminals. A job can consist of one or several independently executing processes that share the resource allocations for that job. Jobs can be grouped into application subsystems that share files and communication channels that are protected from other application subsystems.

Programs and Processes

The four concepts important for understanding how the operating system supports multiprogrammed application systems are the:

- **image**, or executable program
- **process**, or image context and address space
- **job**, or detached process and its subprocesses
- **group**, or set of jobs that can share resources

These concepts are for the most part transparent to the general user whose only contact with the system is the operating system's command language interpreter or an application's command interface. They are, however, significant concepts for the applications programmer. Figure 6-1 illustrates the concepts of groups, jobs, processes, and images.

An image is an executable program. It is created by translating source language modules into object modules, and linking the object modules together. An image is stored in a file on disk. When a user runs an image, the operating system reads the image file into memory to execute the image.

The environment in which an image executes is its context. The complete context of an image not only includes the state of its execution at any one time (known as its hardware context), it also includes the definition of its resource allocation quotas, such as device ownership, file access, and maximum physical memory allocation. These resource allocation quotas are determined by the quotas given to the user who runs the image.

Two or more users can execute the same image concurrently, that is, image code can be shared, in which case the image is executing in two or more different contexts. An image context, including the address space used by an image, is called a **process**. The operating system schedules processes, and a process provides a context in which an image executes.

The distinction between an image and a process is a significant one. We can speak of two processes, each executing the FORTRAN compiler. There may be only one copy in physical memory of the FORTRAN compiler's image, but two different contexts in which the image executes. In one context, the compilation may have just begun, in the other context, it may be almost complete. In one context, the compiler may be reading and writing files listed in one directory, in the other context, the compiler may be reading and writing files listed in another directory.

A process executes only one image at a time, but it provides the context for serially executing any number of different images. For example, when a user logs on the system at an interactive terminal, the operating system creates a process for that user. If the user edits a file, the editor image executes in the context of that user's process. If the user then compiles a program, the compiler image executes in the context of that user's process. A process thus acts as a continuous "envelope" for a user's activities.

An image executing in the context of a process can create **subprocesses**. A subprocess can be thought of as an auxiliary process in which a given image executes. When an image creates a subprocess, it identifies the image to be executed in the context of the subprocess or the source of

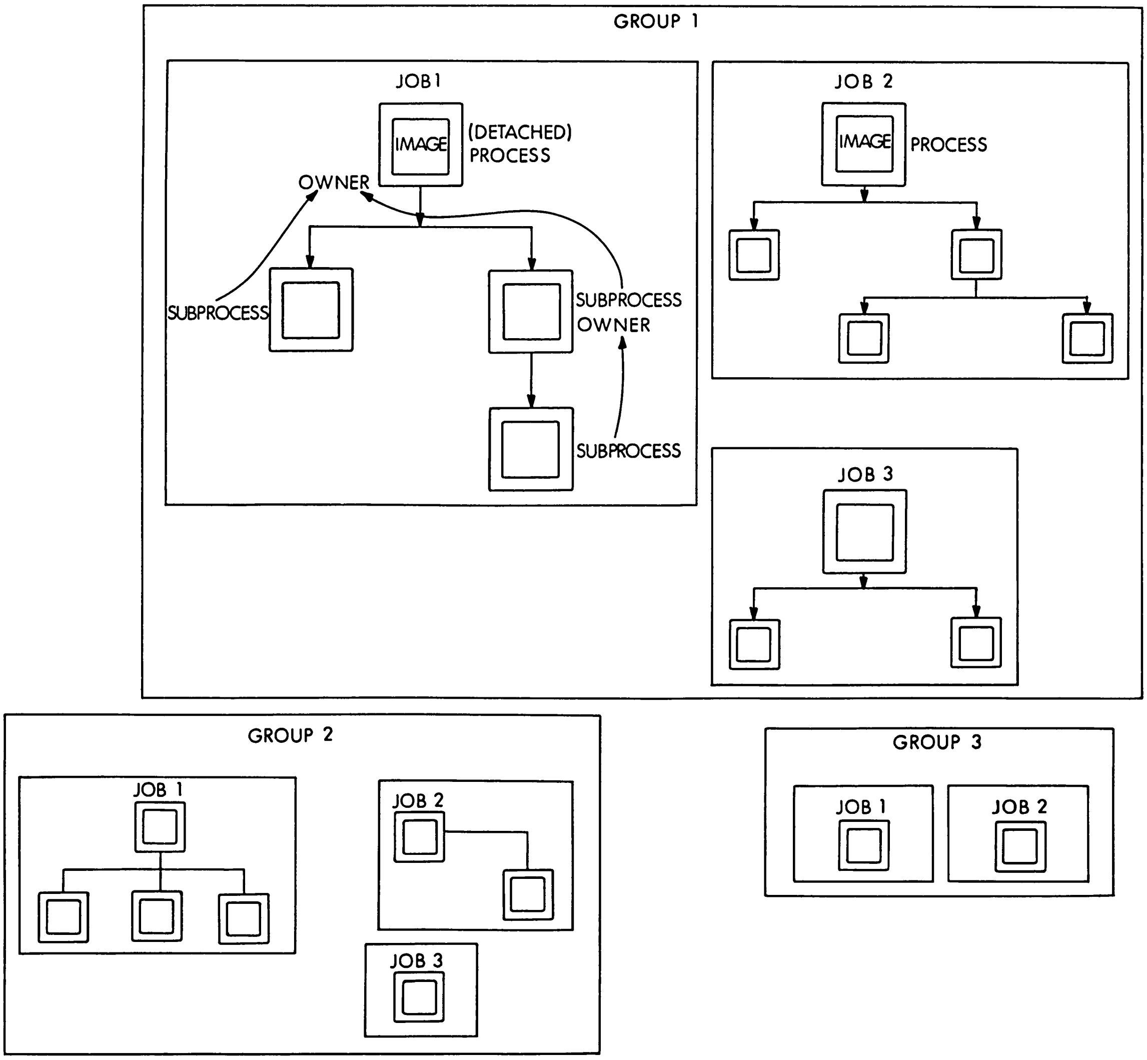


Figure 6-1
Programs and Processes

commands to be interpreted in that process. An image executing in a subprocess context can in turn create other subprocesses.

The process executing the image that creates a subprocess is an **owner process**. An owner process has complete control over the execution of the subprocesses it creates. It determines which of its privileges and how much of its allocated resources it will allow a subprocess to use. It can control the scheduling of its subprocess, and it can delete the subprocess. When an owner process terminates, all of the subprocesses it owns are terminated.

A **detached process** is the process created by the operating system on behalf of a user who logs in to the system and requests services of the system using a command interpreter. A detached process has no owner. Normally, only the operating system can create detached processes, but a suitably privileged application system program could also create a detached process and start up an application command interface to execute images serially in the detached process or any subprocess it creates.

A **job** consists of a detached process and all the subprocesses it creates, and all the subprocesses they create, etc. Jobs are the accounting entities that the system uses to control resource allocation. All processes constituting a job are scheduled independently (they can compete for processor time individually to overlap processing), but they share the total resources allocated to the detached process. The detached process gives a portion of its resource quotas and limits to its subprocesses, which in turn give portions of their quotas to their subprocesses, etc.

Jobs can be associated in **groups**. Groups are the basis for the definition and development of application subsystems. Groups are mutually exclusive, that is, if a job belongs to one group, it does not belong to any other group. A process with appropriate privilege can control the execution of other processes in the same group. Processes in the same group can synchronize their activities using protected group communication facilities.

Resource Allocation

The resources of the system are the processor, memory, and peripherals. The system handles many jobs simultaneously, and each job can have different resource requirements. The operating system enables jobs to share the resources according to their individual needs, and the operating system protects each job and its data from other jobs on the system.

The operating system controls resource allocation dynamically through its scheduling, memory management, device allocation, and I/O processing software, and statically through the authorization of users.

The system manager is responsible for creating an authorization file entry for each user of the system. The authorization file provides the operating system with the resource quotas and limits for each job. For example, there are quotas and limits that control:

- total processor and connect time usage
- number of subprocesses a job can create
- number of simultaneously open files
- process virtual memory usage

Privileges

In addition to providing job quotas, the user authorization file provides the base definition for each user's privileges. There are potentially 64 distinct privileges that can be individually granted or withheld. Among them are privileges that give the job the right to:

- alter the priority of a process
- execute system services which change access mode
- execute operator functions
- create subprocesses
- set up the communication facilities used by cooperating processes
- control other processes in the same group

Whenever the user executes an image, the image can at most acquire only those privileges and quotas granted directly to that user's job by the authorization file, unless the image is a **known image**. Known images are installed by the system manager, and while they execute they provide a second, dynamic set of privileges granted a user.

When the user executes a known image, the process has the privileges and quotas granted to the user in the authorization file, *plus* those run-time execution privileges granted specifically to that image. While that image executes, the user may have the privilege to perform operations not granted when executing any other image. For example, one known image is the operating system's LOGIN image, which enables a user to log on the system. The LOGIN image has the privilege required to access the user authorization file to obtain the user's privileges and quotas.

Protection

The basis for data protection in the VAX-11/780 system is the user identification code (UIC). A UIC consists of two numbers: a group number and a member number. The system manager assigns each user a user identification code (UIC) in the user authorization file. Images that the user executes are given or denied data access privileges based upon the user's UIC.

When a file or an interprocess communication facility is created, it is assigned a UIC and a protection code. The UIC determines which group of users or programs, and which member family within the group, has controlled access to that data. The protection code provides the access control.

The protection code applies to four types of access: **read, write, execute,** and **delete**. Each type of access can be given or denied to:

- the owner: the user or program whose UIC is the same as the UIC assigned to the data
- the group: every user or program whose UIC group number is the same as that assigned the data
- the world: every user or program whose UIC group number is different from that assigned the data
- the system: every user or program whose UIC group number is a system privileged group number (1-8)

For example, in a common application of the protection scheme, a user can create a program image file and assign it the same UIC as the user's own UIC (the default case). The user can give it a protection code to:

- enable the user (and all other users with the same UIC) to read, write, execute, and delete the file
- enable other users in the group to execute the program image, but prevent them from reading, writing or deleting the file
- prevent all users outside the group (other than privileged system users) from reading, writing, executing, or deleting the file
- enable the the privileged system users to read the file (so that it can be backed up, for example)

Read and write access applies to both files and interprocess communication facilities. Delete access applies only to files, and execute access applies only to program image files. (The privileges and quotas granted in the user authorization file control creation and deletion for interprocess communication facilities.)

USER PROCESS ENVIRONMENT

The user program environment is the process, which is the entity the operating system schedules for execution. Each process has its own independent address space in which an image executes. Each image executing in a process can call system service procedures to acquire resources and request special processing services from the operating system. The following paragraphs introduce program virtual address allocation and the fundamental system service procedures available to user programs directly, as well as indirectly through the more complex programmed requests provided by the operating system.

Virtual Address Space Allocation

Process virtual address space is the set of 32-bit addresses that an image executing in the context of a process uses to identify byte locations in virtual memory. For the purpose of allocating virtual memory to processes, the operating system divides process virtual address space into four sets of virtual addresses. The first three sets of addresses are called the program region, the control region, and the system region. The fourth set of addresses is unused.

Figure 6-2 illustrates the general allocation of virtual address space for each process. Addresses in the first two regions are used for process code and data, where the first region is generally used for image specific code and data, and the second for stacks, process permanent data, buffers, and operating system code. Addresses in the system region are also used for code and data maintained by the operating system, but in this case the addresses refer to the same locations in every process context. The system region addresses provide a set of locations whose addresses are independent of process context, and therefore do not have to be context switched.

When a user program is translated and linked, the image is allocated addresses starting with address 512 and continuing up. The first page is not normally allocated (although it can be) because it helps catch programming errors caused by improperly initialized pointers, by branching or jumping to 0, or by passing 0 or other small addresses as arguments. The linker allocates the remainder of address space to image sections according to whether they are shared or private, position independent or position dependent, and read only or read/write such that memory protection can be used to full advantage in preventing and isolating programing errors.

The addresses in the control region are used to identify the locations containing temporary image control information and data such as the stacks, permanent process control information such as I/O channel allocations, and code provided by the operating system. These addresses are allocated from address 2^{31} - 1 *down*. One reason this method of allocating in reverse is convenient is because the control region contains the process stacks, and stacks grow to lower addresses as data are added, and to higher addresses as data are removed.

There are four stack areas reserved in the control region, one for each access mode protection level that the processor provides for software executing in the context of a process. (Refer to the Processor section for a description of access modes.) The stack seen by the image executing in the program region is the user stack. All other stack areas are protected from that image. These stacks are used by operating system software executing in the context of the process on behalf of the image the process is executing. For example, command interpreters use the supervisor stack, and the record management services use the executive stack, and the exception dispatcher and some exception handlers use the kernel stack.

The system region addresses, which start at address 2^{31}, are used to identify the locations containing the entry vectors for system service procedures, followed by locations containing privileged operating system code and data. The system service entry vectors are permanently reserved virtual addresses so that no relinking is required if system services are modified. Other addresses in the system region are not generally used by the image allocated to the program region, and access to areas mapped by these addresses is restricted.

System Services

An image requests services of the operating system directly through calls to the **system services**. The system services are to the operating system what the instruction set is to the processor. They provide all the primary resource request activities, such as I/O processing and interprocess communication. Other programmed requests available to the user are often derived from system services. For example, the record management services use the I/O processing system services as the basis for logical record processing functions.

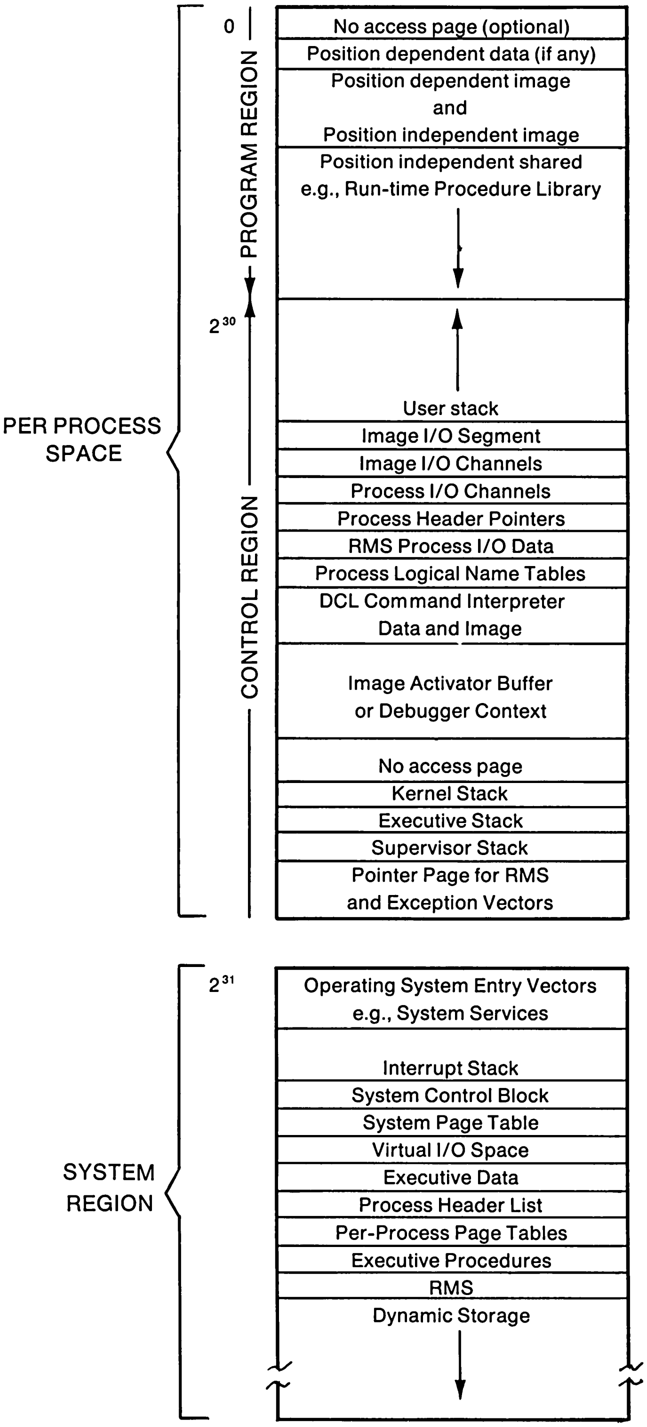


Figure 6-2
Virtual Address Space Allocation

Images that use the system services can be written in assembly language or any native programming language that has a call statement. (Refer to the Languages section and the section on the RSX-11M Programming Environment for system services available in compatibility mode programming languages.) The call interface is the same independent of the programming language selected.

Table 6-1 summarizes the system services available to the applications programmer, some of which are controlled by privilege. The table also lists those system services used primarily by the operating system but which can also be used by suitably privileged application system code.

Table 6-1
System Services

I/O PROCESSING	
Assign I/O Channel ($ASSIGN)	Establish a path for an I/O request.
Deassign I/O Channel ($DASSGN)	Release a path for an I/O request.
Get I/O Channel Device Information ($GTCHAN)	Obtain information about a device that has been assigned.
Create Mailbox ($CREMBX)	Establish a mailbox for the reception of messages and assign it a channel.
Delete Mailbox ($DELMBX)	Mark a permanent mailbox for deletion.
Create Logical Name ($CRELOG)	Assign a logical name to an equivalence name for a process or group.
Delete Logical Name ($DELLOG)	Deassign a logical name.
Translate Logical Name ($TRNLOG)	Search process, then group, then system logical name tables for a logical name and return its equivalence name when the first match is found.
Allocate Device ($ALLOC)	Reserve a device for exclusive use.
Deallocate Device ($DALLOC)	Relinquish an allocated device.
Queue I/O Request ($QIO)	Initiate an input or output operation and continue processing while I/O is in progress.
Queue I/O Request and Wait ($QIOW, $INPUT, $OUTPUT)	Initiate an input or output operation and wait until it is complete before continuing.
Cancel I/O Requests ($CANCEL)	Cancel pending I/O requests on a particular channel.
Broadcast ($BRDCST)	Send a high-priority message to an assigned terminal.
Formatted ASCII Output ($FAO, $FAOL)	Perform ASCII string substitution, and convert numeric data to ASCII representation and substitute in output.
Get Time ($GETTIM)	Obtain the date and time in system format.
Convert Binary Time to Numeric ($NUMTIM)	Convert system format time into integer date and time values.
Convert Binary Time to ASCII ($ASCTIM)	Convert system format time into an ASCII string for date and time.
Convert ASCII String to Binary Time ($BINTIM)	Convert an ASCII string into system format for date and time.
Get Message ($GETERR)	Obtain text of system message from message file.
EVENT FLAG PROCESSING	
Associate Common Event Flag Cluster ($ASCEFC)	Create a process local event flag cluster, or create or associate with a group common event flag cluster.
Disassociate Common Event Flag Cluster ($DACEFC)	Disassociate with a group common event flag cluster.
Delete Common Event Flag Cluster ($DLCEFC)	Mark a group common event flag cluster for deletion.
Set Event Flag ($SETEF)	Turn on an event flag.
Clear Event Flag ($CLREF)	Turn off an event flag.
Read Event Flags ($READEF)	Return the status of flags in a cluster.
Set Timer ($SETIMR)	Set an event flag at an absolute time of day or after a specified interval.
Cancel Timer Requests ($CANTIM)	Cancels all or a selected subset of the Set Timer requests previously issued.
Wait for Single Event Flag ($WAITFR)	Place process in wait state until a particular event flag is set.
Wait for Logical OR of Event Flags ($WFLOR)	Place process in wait state until any one of several event flags is set.
Wait for Logical AND of Event Flags ($WFLAND)	Place process in wait state until all specified event flags in a cluster are set.
ASYNCHRONOUS SYSTEM TRAP PROCESSING	
Set Timer ($SETIMR)	Establish the address of an asynchronous system trap routine to be executed at an absolute time of day or after a given interval of time.
Set Power Recovery AST ($SETPRA)	Establish the address of an asynchronous system trap routine to be executed on a power recovery.
Set AST Enable ($SETAST)	Enable or disable the delivery of asynchronous system traps for the current access mode.
Declare AST ($DCLAST)	Enqueue an asynchronous system trap for delivery to the same or a less privileged access mode.
EXCEPTION CONDITION HANDLING	
Set Exception Vector ($SETEXV)	Establish the address of a process-wide condition handler to be executed when an exception condition occurs.
Set System Service Failure Exception Mode ($SETSFM)	Request generation of a software exception condition when a system service call returns an error.

Table 6-1 (cont)
System Services

Declare Change Mode to User Condition Handler ($DCLCHM)	Establish the address of a condition handler to be executed when a Change Mode to User instruction is trapped.
Unwind Call Stack ($UNWIND)	Delete a specified number of call frames from the call stack following a nonrecoverable exception condition
INTERPROCESS CONTROL	
Create Subprocess ($CREPRC)	Create a subprocess and specify the image that it executes.
Set Process Name ($SETPRN)	Establish a text string to be used to identify a process.
Get Job/Process Parameters ($GETJP)	Obtain information about the current context of a process.
Delete Process ($DELPRC)	Delete this process, a subprocess, or another process in the same group.
Hibernate ($HIBER)	Make this process dormant (but able to receive ASTs) until a wake request.
Schedule Wakeup ($SCHDWK)	Schedule wakeup for this process or a hibernating subprocess at an absolute time of day or after a given interval.
Cancel Wakeup ($CANWAK)	Cancel a scheduled wakeup request for a subprocess, or another process in the group.
Wake Process ($WAKE)	Wake a hibernating subprocess, or another process in the group.
Suspend Process ($SUSPND)	Make a subprocess or another process in a group dormant and unable to receive ASTs.
Resume Process ($RESUME)	Restore the ability of a subprocess or another process in a group to execute.
Exit ($EXIT)	Terminate image execution and return control to the command interpreter.
Force Exit ($FORCEX)	Terminate image execution in a subprocess or another process in a group.
Declare Image Exit Routine ($DCLEXH)	Establish the address of a routine to receive control when an image exits, or add to list of routines.
Cancel Exit Handler ($CANEXH)	Cancel a previously established exit handling routine.
MEMORY MANAGEMENT CONTROL	
Adjust Working Set Limit ($ADJWSL)	Change the maximum number of pages that a process can have in its working set.
Expand Program/Control Region ($EXPREG)	Increase process virtual memory by mapping additional pages at the end of the program or control region virtual address space.
Contract Program/Control Region ($CNTREG)	Decrease process virtual memory by unmapping a given number of pages from the end of the program or control region.
Create Virtual Address Space ($CRETVA)	Map additional pages of virtual addresses into process virtual address space.
Delete Virtual Address Space ($DELTVA)	Unmap pages of virtual addresses from process virtual address space.
Create and Map Private Section ($CRMPSC)	Include a portion of a disk file within process virtual address space.
Create and Map Global Section ($CRMPSC)	Make pages of a file containing sharable code or data accessible to authorized users.
Map Global Section ($MGBLSC)	Include pages of a global section in process virtual memory.
Delete Global Section ($DGBLSC)	Make a global section unavailable for mapping when no more references are made to it.
Set Protection on Pages ($SETPRT)	Change the protection on a given page or range of pages.
Lock Pages in Working Set ($LKWSET)	Prevent a particular page or pages from being paged out of the process working set.
Unlock Pages in Working Set ($ULWSET)	Allow a particular page or pages to be paged out of the process working set.
Lock Page in Memory ($LCKPAG)	Prevent a page or pages from being swapped out of memory.
Unlock Page in Memory ($ULKPAG)	Allow a page or pages to be swapped out of memory.
GENERAL PROCESS STATE CONTROL	
Set Resource Wait Mode ($SETRWM)	Establish whether or not control should be returned immediately when a system service can not be executed because a resource is not available.
Set Process Swap Mode ($SETSWM)	Prevent this process from being (or allow it to be) swapped out of the balance set.
PRIVILEGED OPERATING SYSTEM CONTROL	
Create Permanent Mailbox ($CREMBX)	Establish a permanent mailbox for the reception of messages and assign it a channel.
Delete Permanent Mailbox ($DELMBX)	Delete a permanent mailbox when no more channels are opened to it.
Create System Logical Name ($CRELOG)	Assign a logical name to an equivalence name and place the name in the system logical name table.
Delete System Logical Name ($DELLOG)	Deassign a logical name.
Broadcast ($BRDCST)	Send a high-priority message to all terminals.
Send Message to Accounting Manager ($SNDACC)	Provide accounting information to system accounting routines.
Send Message to Symbiont Manager ($SNDSMB)	Request symbiont manager to perform operation on print file, job queue, or device queue.
Send Message to Operator ($SNDOPR)	Communicate with the system operator.
Send Message to Error Logger ($SNDERR)	Provide information to error logging routines.

Table 6-1 (cont)
System Services

Delete Process ($DELPRC)	Delete any process in the system.
Schedule Wakeup ($SCHDWK)	Schedule a wakeup for any process in the system.
Cancel Wakeup ($CANWAK)	Cancel scheduled wakeup for any process in the system.
Wake Process ($WAKE)	Wake any process in the system.
Force Exit ($FORCEX)	Cause image exit of any process in the system.
Change Mode to Executive ($CMEXEC)	Execute a given routine in executive mode.
Change Mode to Kernel ($CMKRNL)	Execute a given routine in kernel mode.
Adjust Outer Mode Stack Pointer ($ADJSTK)	Modify the current stack pointer for a less privileged access mode.
Declare Change Mode to Supervisor Handler ($DCLCHM)	Establish the address of a condition handler to be invoked when the Change Mode to Supervisor instruction is trapped.
Get Message ($GETMSG)	Return text of system error message from message file.
Put Message ($PUTMSG)	Write a message to the current output and error devices.
Declare Compatibility Mode Handler ($DCLCHM)	Establish the address of a condition handler to be invoked when a compatibility mode fault occurs.
Create Detached Process ($CREPRC)	Create a detached process.
Set Priority ($SETPRI)	Change the execution priority for a process.

I/O System Services

The operating system provides the programmer with two programmed request interfaces for performing input/output operations: the I/O system services and the record management services. The record management services, discussed in the section on Data Management, provide a general purpose file and record programming interface that satisfies most I/O processing needs, and allows the programmer to implement I/O processing quickly. The I/O system services provide the programmer with direct control over the I/O processing resources of the operating system. In particular, the I/O system services enable the programmer to:

- perform both device-independent and device-dependent I/O processing
- read and write blocks on mass storage media using physical (device-oriented), logical (volume-relative), or virtual (file-relative) addressing

The I/O system recognizes several types of device, and within the extents of their capabilities, all devices are programmed in the same manner. All devices can be sequentially accessed, including mass storage devices such as disk and magnetic tape, and record-oriented devices, such as terminals, card readers and line printers. In addition, disk volumes can be accessed randomly.

Mass storage volumes can be either file-structured or non-file structured, according to the choice of the user. The I/O system services enable programmers to use either the physical (device assigned) address or a logical (driver assigned) address for directly addressing blocks on **foreign** mass storage volumes. A foreign volume can be either non-file structured, or structured with the user's own file structure. If the volume is structured using the operating system's Files-11 disk file or ANS magnetic tape structure, the I/O system services enable the programmer to address blocks directly using virtual (file system assigned) addresses.

A special type of record-oriented device is the **mailbox**, which is a virtual device a process creates for the receipt of messages from other processes. Mailboxes are treated like any other record-oriented device: they can be read from and written into using either the I/O system services or record management services. Mailboxes are discussed further in the section on Interprocess Communication.

Before a process requests I/O to a device, it obtains a channel assignment from the operating system. A process can use a device name or a **logical name** in a channel assignment request to identify the device for which the channel is desired.

A device name is a unique name assigned by the operating system to physical devices. The name identifies the type of device and its controller and line or unit number, as applicable. For example, DMA3 is the operating system's device name for the RK06 disk drive unit 3 on controller A, and TTA12 is the operating system's name for the terminal on line 12 on multiplexer A.

A logical device name is any string of characters a user or program assigns to an operating system's device name. The Create Logical Name system service not only enables a process to define logical names for device names, but it enables a process to assign logical names to any portion of a file specification, or to other logical names. Furthermore, logical names can be assigned on a per-process, per-group, or system-wide basis. (For more information on logical names, refer to the section on Data Management Services.)

Once a channel is obtained, a process can issue I/O requests on that channel. The Queue I/O Request system service is a general I/O request interface. All I/O using system services is asynchronous: both I/O and computation can be taking place simultaneously. An I/O request is simply queued to the device driver and control is normally returned to the requesting process before the I/O operation is complete.

The process is responsible for synchronizing with I/O completion. The process can simulate synchronous I/O processing by using the Queue I/O Request and Wait system service, or it can continue to execute during the I/O operation and request I/O completion notification using the general purpose event flag or asynchronous system trap notification mechanisms.

Local Event Flags

An event flag is a status bit used for posting an event, such as I/O completion or elapsed time interval. Event flags are an extremely efficient means of starting up or synchronizing procedures.

Each process has available for its own use two local event flag clusters, each of which contains 32 event flags. Eight flags in the first cluster are reserved by the operating system. A process can set, clear, and read individual event flags, as well as wait for one or more event flags to be set. The advantage to having two clusters of event flags is that the flags in each cluster can be treated as a related group. A process can wait until any of a specified set of flags in a particular cluster is set, or wait until all of a specified set of flags in a particular cluster are set.

Aside from their use with I/O processing and timer scheduling, a process can assign its own meanings to local event flags. Event flags can be used to coordinate several asynchronous events, such as multiple I/O request completions, or to simplify asynchronous processing. For example, a program may wish to know if a terminal user has typed a CTRL/C (indicating the desire to interrupt execution) only at well-defined points during processing. An asynchronous system trap routine can set an event flag to indicate that a CTRL/C has been received.

Asynchronous System Traps

An asynchronous system trap (AST) is a software-simulated interrupt used for event notification within a process. An asynchronous system trap routine is a procedure that handles an AST. AST routines provide an efficient means for processing events that can occur at any time during processing (such as terminal input) because they eliminate the need for polling.

For example, a program can specify AST routines for I/O request processing, timer scheduling, and power recovery. When the I/O operation completes, time interval expires, or power is restored, the operating system declares an AST. When the AST is delivered, the operating system interrupts the process and executes the AST routine. A process can be hibernating and still receive ASTs declared for it.

Code executing at one processor access mode can declare an AST for code executing at the same or a less privileged access mode. The operating system automatically disables AST delivery while an AST routine is executing, and code executing at a given access mode can explicitly disable AST delivery. While ASTs are disabled, the operating system queues any ASTs waiting to be delivered to that access mode in the order in which they were declared. When AST delivery is again enabled, the ASTs are delivered in the order in which they were queued.

Exception Conditions and Condition Handlers

A program may request the processor or a system service to do operations they can not perform correctly. For example, a program might inadvertently issue a divide instruction using a divisor of zero. Normally there is no way to recover and the program can not continue. In this system, however, it is possible for a program to continue if it declares a **condition handler** that can correct the situation. If a user program declares a condition handler, control transfers to the condition handler when an **exception condition** occurs.

This system treats all errors or special events that occur synchronously with respect to a program's execution as exception conditions, and provides a general purpose mechanism for dispatching condition handlers. Exception conditions include:

- errors from which the processor can not normally recover, such as the divide by zero arithmetic trap
- special conditions for which a program does not wish to test continually, for example, the floating point overflow arithmetic trap or unsuccessful system service completion

Some of the exceptions detected by the processor are handled automatically by the operating system. For example, the pager is a condition handler for page faults.

In addition to processor and system service detected exception conditions, any software procedure can define cases for which it will fail or produce an exception by calling a system library procedure that signals an exception condition. The search sequence for a condition handler is independent of the nature of the exception condition: the search sequence is the same whether an exception condition is detected by hardware or software.

A process can declare two kinds of condition handlers: those that are process-wide and those that are applicable to individual procedures. Process-wide condition handlers are declared using the Set Exception Vector system service, which enables a process to declare a primary and a secondary condition handler. Condition handlers applicable to individual procedures are declared by the procedure when it is called using one instruction.

When an exception condition occurs, the exception dispatcher does not differentiate between exception conditions, it simply transfers control to the first condition handler it can find that wants to handle the exception condition. This method for handling exception conditions is an efficient means of transferring control to the appropriate condition handler rapidly, since condition handling is defined by the module or modules in which an exception condition may occur.

For programs written in high-level languages, each language may have different definitions of what is and what is not an exception condition. As the user program calls language functions, the exception conditions for those functions can be handled locally with the procedure. And where exception conditions should be handled on a process-wide basis, the primary and secondary exception vectors provide a top level exception condition trap. For example, when a user program is linked with the debugger, the debugger uses the primary exception vector to declare a process-wide condition handler.

INTERPROCESS COMMUNICATION AND CONTROL

This system supports both simple and complex job definitions. A simple job is a detached process created by the operating system on behalf of the user who logs in at a terminal, or for the purpose of executing a batch job. A simple job serially executes images, but it does not create subprocesses.

A complex job is one in which a detached process creates subprocesses in which designated images execute. These subprocesses can also create their own subprocesses, and so on. The advantage of a complex job over a simple job is that a complex job performs parallel processing operations because it has control over several images executing concurrently.

The following sections describe the services that enable a process to control and communicate with other processes.

Process Control Services

The system services provide process control by enabling a process to:

- create and delete subprocesses
- hibernate then reactivate a process via the Hibernate/Wake and Suspend/Resume system services

The ability to create subprocesses is a privilege granted to a user by the system manager, where the number of subprocesses a job can create is a resource limit. When a process creates a subprocess, it can give the subprocess all or some of its privileges, and it can grant some of its resource quotas and limits to the subprocess. Certain granted quotas are subtracted from the creator's quotas.

The hibernate/wake and suspend/resume mechanisms are methods of process control which are especially efficient in real-time applications. They allow the user to prepare an image for execution and then place it into a wait state until some event occurs which requires its activation.

The Hibernate system service provides the greatest flexibility in sequencing processes for execution. When a Hibernate system service is invoked, normal execution can be resumed only by issuance of a $WAKE system service (or a variant, $SCHDWK, which allows wake-up at an absolute time or at a fixed time interval). However, a hibernating process can be interrupted temporarily by the delivery of an AST (Asynchronous System Trap). When the AST service routine completes execution, the process continues hibernation. If, however, the the process calls the $WAKE system service during execution of the AST service routine, the process wakes itself after the service routine completes. Figure 6-3 shows an example of a program which uses the hibernate and wake system services.

```
Process: GEMINI

ORION: DESCRIPTOR <ORION>                 ;SUBPROCESS NAME
FASTCOMP: DESCRIPTOR <COMPUTE.EXE>        ;IMAGE

             .
             .
             .
1          $CREPRC_S PRCNAM=ORION,
                     IMAGE=FASTCOMP       ;CREATE ORION - HE'LL
                                          ;SLEEP
           BLBC      R0,ERROR             ;BRANCH IF SERVICE ERROR
                                          ;CONTINUE
             .
             .
             .
3          $WAKE_S   PRCNAM=ORION         ;WAKE ORION
           BLBC      R0,ERROR             ;BRANCH IF SERVICE ERROR
             .
             .
             .
           $WAKE_S   PRCNAM=ORION         ;WAKE ORION AGAIN
           BLBC      R0,ERROR             ;BRANCH IF SERVICE ERROR

Process: ORION

FASTCOMP:
2          .WORD     0                    ;ENTRY MASK
10$        $HIBER_S                       ;SLEEP 'TIL GEMINI WAKES
                                          ;ME
           BLBC      R0,ERROR             ;BRANCH IF SERVICE ERROR
             .                            ;PERFORM...
             .
             .
           BRB       10$                  ;BACK TO SLEEP
```

Notes:

1. Process GEMINI creates the process ORION, specifying the image name FASTCOMP.

2. The image FASTCOMP is initialized, and ORION issues the $HIBER system service.

3. At an appropriate time, GEMINI issues a $WAKE request for ORION. ORION continues execution following the $HIBER service call. When it finishes its job, it loops back to repeat the $HIBER call and to wait for another wake.

Figure 6-3
Program using Hibernate/Wake System Services

Using the $SUSPEND system service, a process can place itself or another process into a wait state similar to hibernation. However, a suspended process can not be as easily activated as a hibernating one. It cannot, for example, be interrupted by delivery of an AST. Nor can it wake itself but can resume normal execution only following issuance of a $RESUME system service by another process.

The interprocess system services can be used by a process to control another process executing in the same group. While only an owner process can create and delete subprocesses, a process can be given the privilege to suspend, resume, and wake other processes in its group.

Jobs with sufficient privilege can also create detached processes, and delete, suspend, resume, or wake any process in the system. These privileges are normally reserved for the operating system.

Interprocess Communication Facilities

In addition to providing process control services, the operating system provides process communication facilities for synchronizing execution, for sending messages, and sharing common data. The three techniques that cooperating processes can use to communicate are:

- common event flags
- mailboxes
- shared areas of memory

Common event flags are available by group association to processes within jobs. Mailboxes and shared areas of memory are more general purpose facilities which can be limited or unlimited in scope. They can be limited to a specific member family within a group or to a specific group of jobs, or they can be extended to all jobs in the system.

Common Event Flags

In addition to the local event flags available to each process, cooperating processes can communicate using common event flags. Every group in the system can define any number of common event flag clusters. Each cluster contains 32 flags. The flags can be assigned any meaning for the processes in the group.

Each process in a group can associate with up to two of its group's common event flag clusters at one time. A process can read, set, clear, or wait for common event flags to be set. The ability to read, set, or clear event flags is controlled by the protection code and User Identification Code assigned to the common event flag cluster.

Mailboxes

A mailbox is a record-oriented virtual I/O device created by a process. Mailboxes can be used to pass status information, return codes, messages, or any other data from one process to another. When a process creates a mailbox, it limits the size of the messages it can receive and the total amount of buffer space available for the mailbox and its messages. A process can also protect its mailboxes from read and/or write access by any process outside its member family or outside its group.

All of the I/O system services and record management services can be applied to mailboxes. Other processes write messages to a process' mailbox by queuing write requests for the device. A process reads messages in its mailbox by queuing read requests for the device. A process can request AST notification when anything is written to its mailboxes, and it can assign mailboxes logical names.

Shared Areas of Memory

The system supports a high degree of code and data sharing through the use of global sections. A global section is a copy of all or a portion of an image or data file that can be mapped in a process virtual address space at run time. Global sections can be used for shared data structures, as communication regions for cooperating processes, or they can be used simply to eliminate multiple copies of image code or data.

Global sections can be created dynamically by a process or they can be permanently present in the system. Dynamically created global sections are mapped into processes that reference them, and deleted when no more references are made to them. Permanent global sections are created from **sharable images** by the system manager. They are loaded into and removed from memory dynamically as references are made to them.

Each process that maps a global section into its virtual address space can have a different access privilege to a global section. When a global section is created, it is assigned a User Identification Code (UIC) identifying the group and member family to which the global section belongs, and a protection code identifying the read and write access privileges of processes in the system. Global sections can be shared by all processes in the system, or shared only by processes within a particular group, or shared only by processes within a particular job. One or more controlling processes can have write privileges while other processes in the system, group, or job have only read privileges.

A process can map to a global section explicitly by issuing a Map Global Section system service, or it can be mapped implicitly by referring to a permanent global section. If an image references a permanent global section, the linker does not normally include the global section in the image. When the image is executed, the image activator calls the Map Global Section system service on behalf of the image. For example, the Common Run-Time Procedure Library is a permanent global section implicitly mapped into images that reference library procedures. The use of permanent global sections significantly reduces the size of programs using common library procedures and the overall system memory requirements.

MEMORY MANAGEMENT

In a multiprogramming system, many processes coexist in memory to make the most use of the processor: the system can switch between process contexts without incurring disk I/O to bring processes in and out of memory. In most multiprogramming environments, however, the number, size, and kind of concurrently executing processes changes rapidly, while the amount of memory available for processes remains constant. Users log on and off the system, production activities vary periodically, and special production jobs occur. Since it is generally inefficient to have available the maximum amount of memory that might ever be needed at one time, it becomes the task of the operating system to provide a dynamic memory that responds to the changing multiprogramming environment.

This system uses two interdependent complementary techniques to allocate limited memory to competing processes: paging and swapping. These techniques relieve the general programmer of concern for memory allocation while still allowing system programmers to optimize program performance in limited configurations. This section and the following section on scheduling discuss how this system's paging and swapping techniques extend limited memory resources with minimum effect on the system or programs when the system has sufficient memory to hold all concurrently executing processes.

Mapping Processes into Memory

The operating system's memory management software is responsible for creating and maintaining the information used to map the virtual addresses used in a program to physical memory addresses. The unit mapped is the **page**, which is a block of 512 contiguous byte locations in physical memory.

Virtual addresses are also grouped into 512-byte pages, and each page of virtual addresses can be mapped to a page of real memory locations. Any number of virtual pages can be mapped to one physical page. Unlike systems that partition or statically allocate portions of physical memory, this system dynamically allocates physical memory, with the result that pages of a process may be scattered anywhere throughout memory. It is never the concern of the programmer to determine how physical memory is allocated. To illustrate this, Figure 6-4 shows how two processes might be mapped into physical memory.

When a process is created, the operating system sets up its mapping information, called **page tables**. Each process has its own page tables mapped by system region virtual addresses. (Refer to the Processor section on memory management for a complete description.) Initially, a process page table simply maps those pages of the control region that define the permanent process context.

When a program is linked, the addresses the linker assigns in the image are always virtual addresses. The linker has no knowledge of how physical memory is allocated. Its primary function is to build descriptions of the size and protection requirements of the program's code and data areas.

When an image is executed, the memory management software uses the linker's descriptions of code and data areas to map image virtual pages to physical pages. The operating system's image activator builds the image's mapping information in the process' page tables.

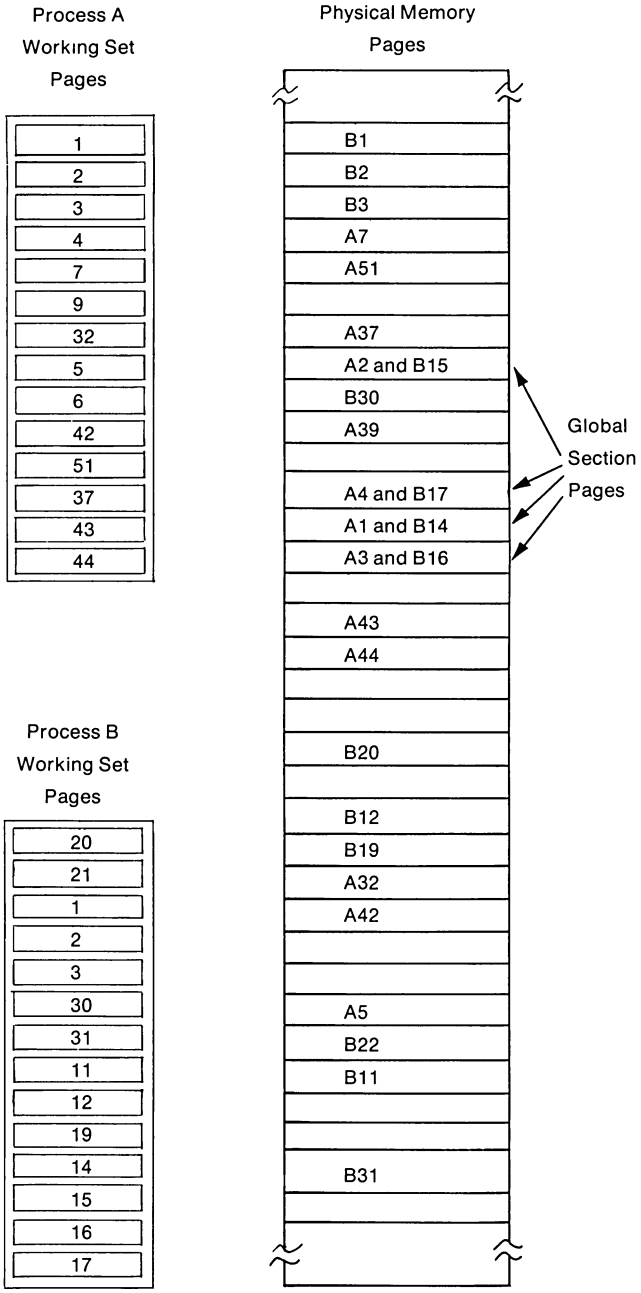


Figure 6-4
Mapping Processes into Memory

Process Virtual Memory and Working Set

The total virtual memory requirement of a process is called **process virtual memory**. Process virtual memory consists of all the pages of the process program region and control region which are mapped by the process page tables.

At any one time, some of the pages of process virtual memory may be mapped to disk and some to physical memory. The physical memory requirement of a process is the process **working set**. When a process is executing, a process working set consists of all the pages of a process' virtual memory residing in physical memory that the process can access directly without incurring a page fault, *plus* any actively used portions of the process page tables and process header information.

The working set is a dynamic characteristic of a process that has both minimum and maximum size limits. The system designates a required minimum number of pages that has to be in a process working set, and the system manager defines the maximum number of pages allowed in any

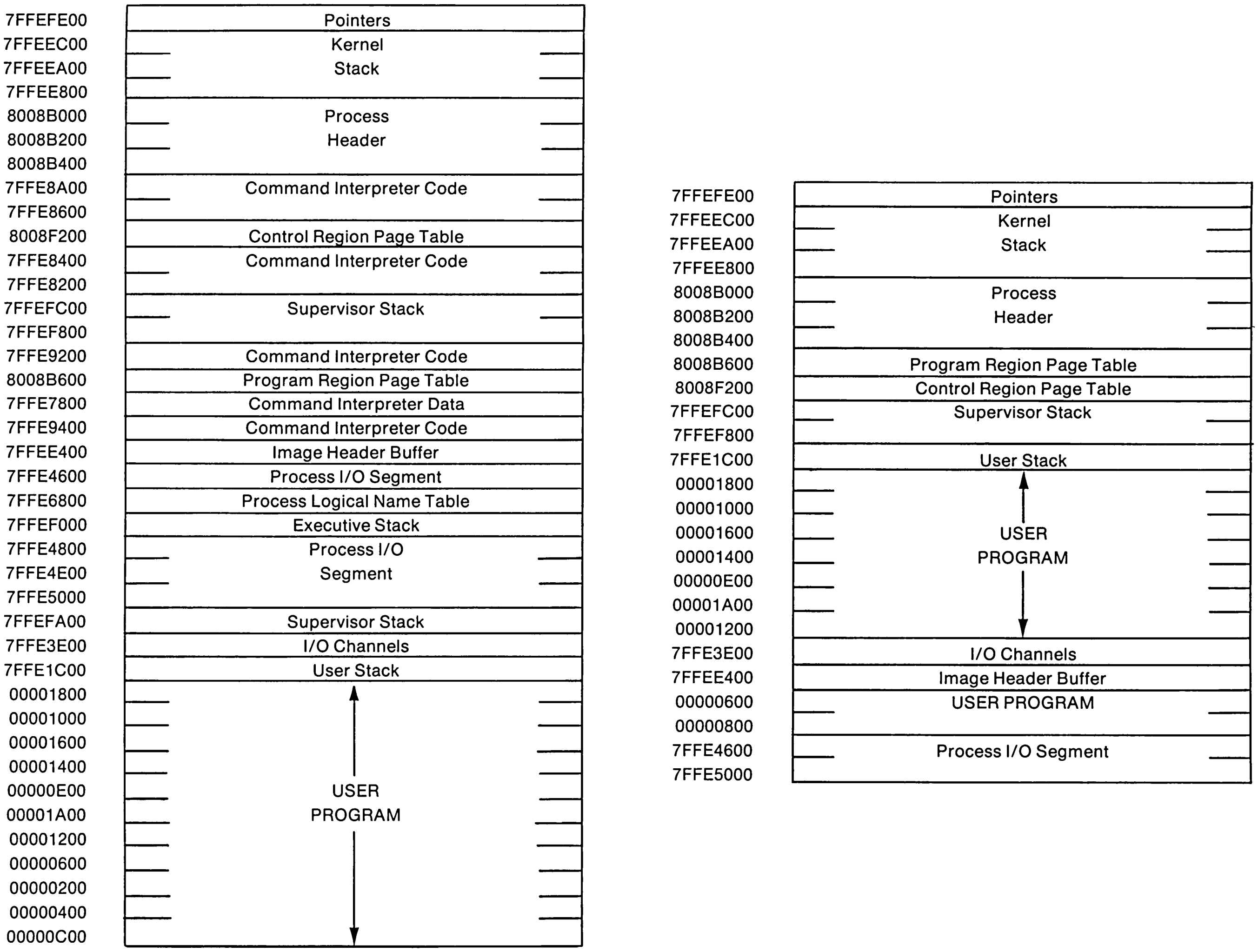


Figure 6-5
Process Working Sets

one job's working set in the user authorization file. The size of a process working set affects its paging and swapping performance, as well as affecting the number of process working sets that can be resident when the process working set is resident.

Paging

Through its paging technique, the operating system can execute programs that are too large to fit in the amount of physical memory allocated to a process, without requiring the programmer to define overlays. Inactive portions of a program are automatically stored on disk while the active portions are resident in memory. When the program references a disk-resident portion of the program, the operating system reads in, or **pages**, the referenced portion, moving out other portions of the program to disk if necessary. This system's paging technique has several features that distinguish it from other techniques:

- clustering, or the ability to read in several pages at one time
- paging processes against themselves, and not the system
- maintaining an available page pool from which processes can recover recently discarded pages without incurring disk I/O
- writing out only the modified pages released from a process working set
- activating a process waiting for page fault I/O for the purpose of executing AST routines when they are delivered

When the operating system activates an image for the first time, a number of pages are read into memory from the image file on disk. The number of pages read in the first time can be controlled by a cluster factor the programmer assigns to the image. The ability to read in several pages at once allows the image to execute for some time without incurring page faults.

A process is subsequently paged only when it executes an image that needs more pages than the process is allowed to have in its working set. If the number of pages in the image plus the number of pages for the remainder of the process is less than the working set size limit, all the pages are read in and the process is never paged.

If all the pages are not read in initially, at some point the image is going to reference the pages that have not been read in. At that time, the process incurs a page fault, that is, a reference to a page not mapped in the process working set.

The operating system's pager is a condition handler that executes when a process incurs a page fault. If the working set size limit has not yet been reached, the pager reads in the faulted page from disk, plus any additional pages, again according to a cluster factor specified by the programmer for that section of the image.

If a page fault occurs when the working set size limit is reached, the pager obtains a page from a pool of available pages to read in the faulted page, and releases the least recently faulted page from the process working set into the pool. Figure 6-5 illustrates two different size working sets for a process running the same program in each case. The illustration shows the order in which the pages were faulted. (Refer to Figure 6-2 for a guide to how the pages appear in virtual address space.)

The pager pages a process only against itself. It does not release pages of one process to satisfy another's needs. This ensures that only those processes that need paging are affected by paging. Other processes in the system need not be affected by another process' memory requirements.

The pool of available pages works as a cache of pages that effectively extends a working set size above its limit when few processes are competing for memory resources and there are many pages in the pool. If a process faults a page that was released and is still in the pool, the page does not have to be read in from disk, it is simply remapped into the working set.

When a page is released, it is placed on one of two lists: the free page list or the modified page list. Modified pages are pages the process has written into and, if they need to be added to the free page list to be used by another process, must be saved on disk. Modified pages are only written to disk when the processor is idle or when the modified page list exceeds a threshold size.

Virtual Memory Programming

The processor provides the programmer with a large virtual address space and rapid address translation, and the operating system provides the programmer with extremely efficient mapping and paging algorithms. Furthermore, these memory management mechanisms are totally transparent to the application programmer. It is not necessary for a programmer to be concerned with address allocation or page mapping: the high-level language compilers and the linker take advantage of the memory management mechanisms to set up the memory allocation optimal for most programming requirements.

For those systems with limited memory or special processing requirements, however, this system enables users to control and optimize memory management. The system manager can control the memory allocation requirements of the system as a whole by initialization parameters such the desired and minimum acceptable number of available pages, and of individual jobs by user authorization parameters such as paging file usage limit and maximum working set size. It is possible to have a process avoid paging entirely by making its working set size equal to its virtual memory requirements, or to reduce paging by choosing a working set size that satisfies the average demand for pages over time.

The programmer also has the ability to control memory allocation for images in two ways: through properly coded programs and through the memory management system services. This system's memory management software optimizes for program locality. Programs that incur paging infrequently are those in which the code and data used during each stage of processing are contained in the fewest possible number of virtually contiguous pages.

For the most part, the linker allocates virtual addresses so that images require the minimum mapping information possible. The programmer can also ensure that images that process large data structures require the least possible mapping information and potential paging by organizing data structures as if they were disk-resident files. In general, the programmer need not be concerned with data structures such as tables and arrays whose elements are virtually contiguous and sequentially processed.

Large data structures that are randomly accessed can, however, be optimized. For example, processing down a linked chain in which the chain elements are spaced far apart with no useful data in between requires that an image reference a large number of pages in a short period of time. If all of the pages cannot fit in the process working set at the same time, the references to successive chain elements will incur disk I/O.

On the other hand, a large data structure can be efficiently accessed using directory trees, where a page or set of consecutive virtual pages contains all one kind of information. One page can contain all of the information that points to randomly arranged, but virtually contiguous, pages containing the data processed at that locality.

Programmed Memory Management Services

For those who have special processing requirements, there are programmed system services that control memory management within the quotas and limits assigned by the system manager. These memory management system services enable a process to:

- modify the working set size limit
- add or delete pages from process virtual memory
- expand or contract the program region or control region
- lock pages in the working set
- lock pages in physical memory

A program can impose a limit on process working set size anywhere between the minimum required by the system and the maximum specified by the system manager. The limit can be adjusted in accordance with program behavior and time-critical requirements. By maintaining the smallest working set size consistent with an acceptable paging rate, a program that temporarily requires a large working set can reduce its impact on the system. For example, a process control program or simulator might use a small working set while processing interactive initialization commands. Once time-critical processing is underway, the program can expand its process working set size to reduce paging. When time-critical processing is finished, the program can contract the working set.

A process can add selected pages to and delete selected pages from its virtual memory dynamically. Deleting a page is in effect saying that the image is no longer going to use those virtual addresses, and the operating system does not need to map them to pages in virtual memory. Deleting read/write pages (such as those used for interprocess communication) as soon as they are no longer used eliminates the need for the system to write them out as modified pages to a paging file. When an image has reached its paging file quota, it can delete pages in order to map other pages in its virtual address space.

A process can request an extension to the amount of virtual memory allocated to its program region. The operating system will map zero-filled pages into the process virtual address space following the highest addressed page allocated for the program region. This service is useful for dynamically creating data arrays whose size is not known beforehand, and it eliminates the need for allocating a data area in a program image. A process can also extend the initial allocation of pages for the user stack by requesting the operating system to map zero-filled pages into process virtual address space preceding the lowest addressed page allocated for the control region.

In unusual situations, a process can lock pages in its working set. Locking a page in the working set is useful when a process does not reference a particular page regularly, but the page needs to be in the working set to increase the performance of the code in that page. For example, it might be desirable to keep the page containing asynchronous system trap routines in a working set to ensure that the routines are started up rapidly when an AST is delivered. Note, however, that locking a page in the working set causes other pages to be paged more frequently, since the page will not be paged out, no matter how long it has been in the working set. A page can be unlocked when it is no longer necessary to keep it in the working set.

It is also possible to lock pages in memory. A page locked in memory is not only locked in the working set, it is not swapped out with the process. This service is useful for time-critical processes that need to keep buffers in memory for I/O transfers.

PROCESS SCHEDULING

VAX/VMS features event-driven scheduling based on process priority. Unlike traditional timeshared scheduling systems, this system's ability to respond to events enables it to dispatch real-time processes efficiently as well as to share processing time among normal processes competing for resources. Furthermore, priority assignment enables the user to bias processor time allocation based on process activity, to bias the allocation absolutely for certain processes, or to mix both allocation methods.

The operating system's scheduler and swapper are responsible for ensuring that the processes executing in the system receive processor time commensurate with their priority, which is controlled by assignment, and with their ability to execute, which is controlled by system events.

System Events and Process States

In VAX/VMS, dispatching a process for execution involves little decision making. The selected process is always the highest priority executable process. The real scheduling decisions are made as the result of system events that make processes executable.

A system event is an event that affects the ability of a process in the system to execute. System events include events external to the process currently executing, such as I/O completion or timer interrupt. System events also include events internal to the process currently executing. The process may issue a wait request or a hibernate request, or it may request or release a system resource, for example, a page of memory.

Every active process in the system is listed in one of several state queues that identifies whether or not a process is executable, and if not, the event or resource for which the process is waiting. Whenever a system event occurs, the scheduler adjusts the process state queues accordingly. For example, the scheduler adds a process to the executable state queue when a resource for which it is waiting becomes available, or removes it when it requests to wait for an event or resource.

The executable state queue supplies the scheduler with a list of processes that are eligible to execute. Priority determines which process among those eligible executes. Rescheduling occurs when a system event makes executable a process with higher priority than the one currently executing.

Unlike timeshared scheduling, therefore, event-driven scheduling is based on the activities of the processes themselves, not on a time limit imposed by the scheduler. Because scheduling intervals are determined by system events, the interval between rescheduling is random. Quantum keeping and requested timer events provide a minimum level of event activity but, in practice, the average interval between events is determined by the duration of the typical I/O operation.

Priority: Real-Time and Normal Processes

The scheduler recognizes 32 scheduling priorities, where priority 31 is high and 0 is low. Priorities 31-16 are for real-time processes, and priorities 15-0 are for normal processes. When a process is created, the system assigns it a scheduling priority. A program image the process executes can modify the process priority using a system service. The system manager grants jobs the privilege to execute at real-time priorities.

The scheduler maintains a queue for each scheduling priority. processes having the same priority are listed in the same queue. The priority assigned to a process when it is created is its base priority. The scheduler does not alter the priority of a real-time process during execution. The scheduler may temporarily increase the priority of a normal process during its execution, but its priority never drops below its base priority.

Scheduling by strict priority for real-time processes and by potentially modifying priority for normal processes allows the scheduler to achieve maximum overlap of compute and I/O activities while still remaining responsive to high-priority real-time applications.

Scheduling Real-Time Processes

When a system event occurs that makes a real-time process eligible to execute, it receives control of the processor unless another higher priority process is currently executing. A real-time process retains control of the processor until it finishes execution, enters a wait state, or is preempted by a higher priority process. (Note that under VAX/VMS, real-time processes actually have a higher priority than system processes, thus ensuring that real-time processing will never be encumbered by system overhead.)

A higher priority real-time process can preempt any lower priority process whenever a system event occurs that makes it eligible to execute. For example, a device interrupt may occur that signals the completion of an I/O transfer requested by the higher priority real-time process.

When a real-time process is preempted to dispatch a process of higher priority, the preempted process is placed at the end of its priority queue. This rotates processes within a priority, with the result that available processor time is distributed among processes of the same priority.

Scheduling Normal Processes

When no real-time processes are executing, the scheduler distributes processor time among the processes on the normal priority levels. As with real-time processes, the scheduler selects the highest priority ready-to-execute normal process. That process executes until it finishes execution, enters a wait state, or is preempted by a higher priority process. Unlike real-time process scheduling, however, the scheduler modifies normal process priority whenever a system event occurs for a normal process and whenever a normal process is scheduled.

When a system event occurs that affects a normal process, the scheduler increases the priority of the normal process (but not to more than the maximum priority of 15) and places the process at the tail of the queue for its new priority. The amount of priority increment depends on the nature of the event. For example, the scheduler increases the priority of a normal process on the following events:

- terminal input completed
- terminal output completed
- resource available
- wake, resume, delete request received
- non-terminal I/O completion, page fault completion, or other event

The priority increments assigned to each type of event can be modified by the system manager. In this case, the terminal I/O events receive the highest priority increments to enable the system to be most responsive to the interactive terminal user. When the scheduler increases a normal process' priority, that process gets control of the processor if its new priority is higher than that of the process currently executing.

Each time a normal process is scheduled, the scheduler decreases its priority by one (unless it is already in its base priority queue) and places it at the end of that priority queue. The effect of dynamically increasing and decreasing normal process priority ensures maximum overlap of computation and I/O.

Swapping and the Balance Set

It is the job of the swapper to keep the scheduler supplied with the highest priority executable processes in configurations that do not have a sufficient amount of physical memory to keep all process working sets memory-resident. The balance set is the set of all process working sets that are currently in memory. The swapper ensures that the balance set always contains the highest priority executable processes by moving low priority or nonexecutable memory resident process working sets to a swap area on disk, and moving high priority or executable process working sets into memory.

Swapping is a very efficient way of extending limited memory resources when many processes are executing concurrently. process working sets for small processes (less than 64K bytes or 128 pages) can be swapped in and out of memory in one disk I/O operation. Where paging extends limited memory resources on a per-process basis and is limited to moving few pages in and out of memory, swapping balances the memory requirements of the system as a whole.

The swapper is activated whenever a system event occurs that can make a nonresident process resident, a nonresident process executable, or a resident process nonexecutable. For example, a resident process might release sufficient memory to enable the swapper to move in a nonresident process. An I/O completion event might make a nonresident process executable. A resident process might enter a wait state and become nonexecutable. In any case, the swapper uses three conditions to determine which processes should be swapped in and which should be swapped out:

- which processes are executable and which are not (and the reason for the wait state)
- what the process priorities are
- whether a process balance set quantum has expired

The balance set quantum effectively enforces a swapping rotation for compute-bound normal processes. Every normal process is assigned a time quantum that provides a guaranteed minimum amount of time in which the process can perform useful work before it is eligible to be swapped out of the balance set. A process can be preempted many times before it has received its full quantum. It remains in the balance set until it completes its first quantum unless a real-time process that is swapped out becomes executable and no other processes can be swapped out to make room for the real-time process.

Programmed process Control Services

In addition to the programmed system services that enable processes to create, delete, suspend, resume, and wake other processes, or to hibernate and wake themselves, a process can control the manner in which it is scheduled by:

- setting process swap mode
- setting resource wait mode

A suitably privileged process can request that it not be swapped out of the balance set, even when it becomes inactive. This is useful for high priority real-time processes that need to be activated rapidly when they become executable.

Normally, when a process requires dynamic resources of the system and they are not available, the process enters a wait state until the resources become available. Dynamic resources primarily include the buffer space needed for mailboxes, I/O requests, etc. A process can request to be notified when resources are not available and continue executing instead of entering a wait state.

I/O PROCESSING

The I/O processing system consists of several highly modular, interdependent components that enable programmers to choose the programming interface and processing method appropriate for their needs, without incurring run-time space or performance overhead for features not used. In addition, the I/O request processing software takes advantage of the hardware's ability to overlap I/O transfers with computation, switch contexts rapidly, and generate interrupts on multiple priority levels to ensure the maximum possible data throughput and interrupt response. Figure 6-6 presents an overview of the major I/O processing system components and their relationships.

Programming Interfaces

The I/O programming tools are the record management system VAX-11 RMS, for general purpose file and record processing, and the Queue I/O system services, for direct I/O processing. Table 6-2 summarizes the programming interfaces.

RMS (discussed in the section on Data Management Services) provides device-independent access to file-structured I/O devices. The most general purpose type of access enables programs to process logical records; RMS automatically provides record blocking and unblocking.

RMS users can also choose to perform their own record blocking on file-structured volumes such as disk and magnetic tape, either to control buffer allocation or optimize special record processing. Users performing their own record blocking address blocks using a virtual block number (which is the number of the block relative to the file being processed) for volume-independent processing.

The I/O system services provide both device-independent and device-dependent programming. Users perform their own record blocking on file-structured and non-file structured devices. Virtual block addressing is used on Files-11 disk or ANSI magnetic tape volumes. In addition, users with sufficient privilege can perform I/O operations using either logical or physical block addressing for defining their own file structures and accessing methods on disk and magnetic tape volumes.

Ancillary Control Processes

Both RMS and the I/O system services use the same I/O control processes, called ancillary control processes (ACPs), for processing file-structured I/O requests. An ACP provides file structuring and volume access control for a particular type of device. Typical ACP functions

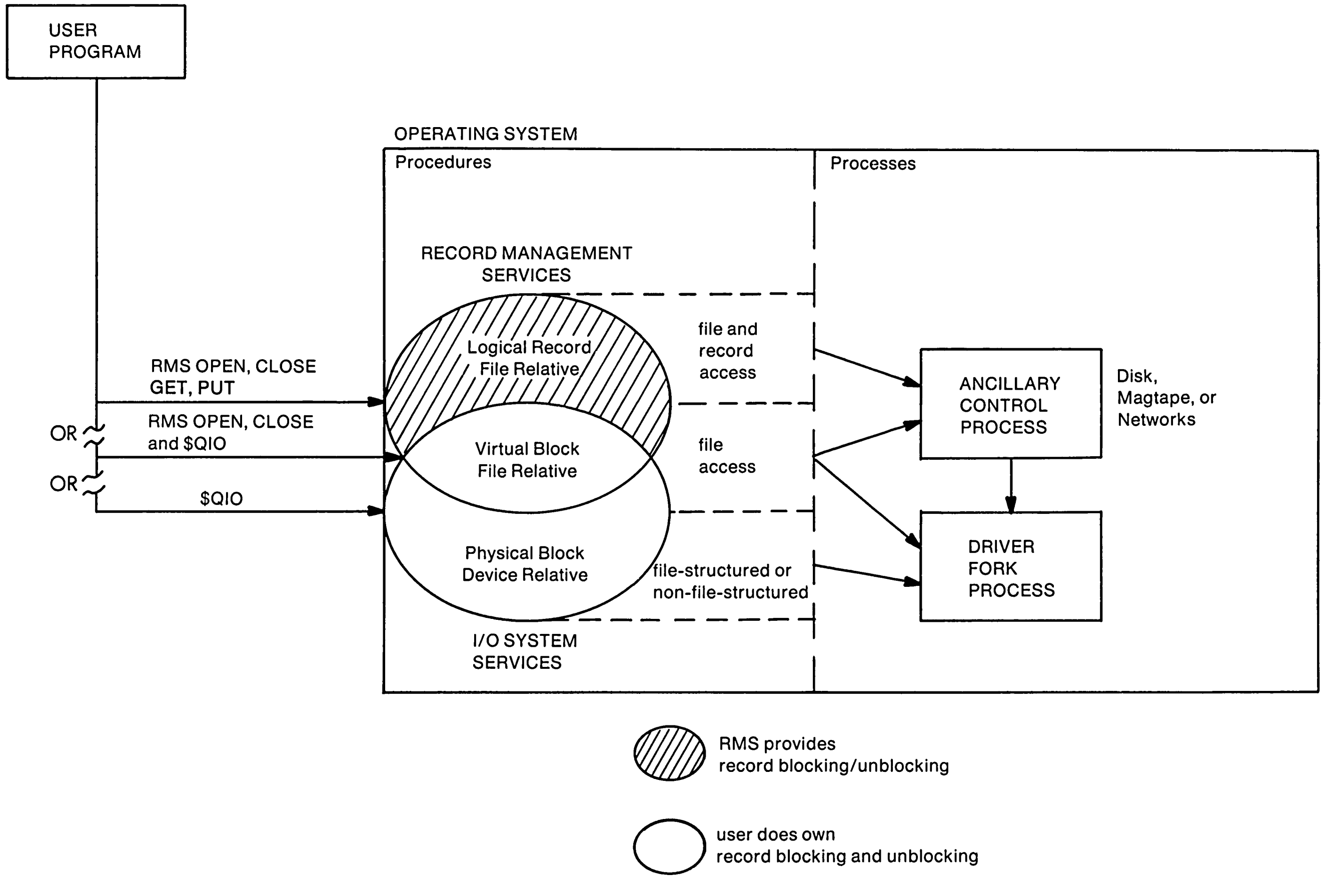

USER PROGRAM
OPERATING SYSTEM
Procedures
Processes
RECORD MANAGEMENT SERVICES
RMS OPEN, CLOSE GET, PUT
OR
RMS OPEN, CLOSE and $QIO
OR
$QIO
Logical Record File Relative
Virtual Block File Relative
Physical Block Device Relative
file and record access
file access
file-structured or non-file-structured
I/O SYSTEM SERVICES
ANCILLARY CONTROL PROCESS
Disk, Magtape, or Networks
DRIVER FORK PROCESS
RMS provides record blocking/unblocking
user does own record blocking and unblocking

Figure 6-6
User Interfaces to I/O Services

Table 6-2
I/O Processing Interfaces

METHOD	PROGRAM INTERFACE	I/O COMPONENTS	PURPOSE
Record I/O	RMS requests	RMS, ACP and Driver	Use Files-11 disk or ANS magtape file structure, use RMS record access methods
File I/O	RMS OPEN and $QIO requests	RMS for OPEN, ACP and Driver	Use Files-11 disk or ANS magtape file structure, implement own record access methods
Device I/O	$QIO requests	Driver	Fast dumps to disk or magnetic tape, foreign file structure

would include creating a directory entry or file, accessing or deaccessing a file, modifying file attributes, and deleting a directory entry or file header. There are three kinds of ACPs provided in the system: Files-11 disk, ANS magnetic tape, and network communications link.

The RMS and I/O system services programming interfaces are the same regardless of the ACP involved, but since ACPs are particular to a device type, they do not have to be present in the system if the device is not present. There is one network ACP process for all DECnet network communications links in the system, and none if the system is not in a network.

Device Drivers

Once the ACP sets up the information for file-structured I/O requests, a request can be passed on to a device driver. All non-file structured I/O requests are passed directly to a device driver.

A VAX/VMS driver performs the following functions:

- Defines the peripheral device for the rest of the VAX/VMS operating system.
- Defines the driver for the operating system procedure that maps and loads the driver and its device data base into system virtual memory.
- Initializes the device (and/or its controller) at system startup time and after a power failure.
- Translates software requests for I/O operations into device-specific commands.
- Activates the device.
- Responds to hardware interrupts generated by the device.
- Reports device errors.
- Returns data and status from the device to software.

Device drivers work in conjunction with the VAX/VMS operating system. The operating system performs all I/O processing that is unaffected by the particular specifications of the target device (i.e, device-independent) processing. When details of an I/O operation need to be translated into terms recognizable by a specific type of device, the operating system transfers control to a device driver (i.e., device-dependent processing). Since different peripheral devices expect different commands and setups, each type of device on a VAX/VMS requires its own supporting driver.

The VAX/VMS operating system contains device drivers for a number of standard DIGITAL-supported devices. These include both MASSBUS and UNIBUS devices. In addition, the user can write additional drivers for non-standard UNIBUS devices. The manual entitled *VAX/VMS Guide to Writing an I/O Driver* provides detailed information on writing, loading and debugging drivers.

I/O Request Processing

All I/O requests are generated by a Queue I/O (QIO) Request system service. If a program requests RMS procedures, RMS issues the Queue I/O Request system service on the program's behalf. Queue I/O Request processing is extremely rapid because the system can:

- keep each device unit as busy as possible by minimizing the code that must be executed to initiate requests and post request completion
- keep each disk controller as busy as possible by overlapping seeks with I/O transfers

The processor's many interrupt priority levels increase interrupt response because they enable the software to have the minimum amount of code executing at high priority levels by using low priority levels for code handling request verification and completion notification. In addition, device drivers take advantage of the processor's ability to overlap execution with I/O by enabling processes to execute between the initiation of a request and its completion. User processes can queue requests to a driver at any time, and the driver immediately initiates the next request in its queue upon receiving an I/O completion interrupt.

All access validation and checking takes place before an I/O request is actually queued. For file-structured I/O requests, the Queue I/O Request system service obtains all the virtual block mapping and volume access checking information from the ACP. For example, on virtual block I/O requests for multivolume files, the system service obtains from the ACP the mapping information that enables it to queue requests to different drivers when the user's I/O request involves a transfer that spans volumes. The Queue I/O Request system service also checks the validity of the function requested (read, write, rewind, etc.) for the particular device. Because all access validation and function checking is performed before the request is queued, the driver has little to do to initiate a request.

Once the system service has verified the I/O request, it raises the interrupt priority level to that of the driver. The only activity it has to perform at this level is a test to see if the driver is busy. If the driver is not busy, it calls the driver. Otherwise, it queues the request according to the priority of the requesting process and immediately returns to the user process.

When the driver is called, it initiates the request and returns to the user process. Because disk seeks do not require the controller once they are initiated, if a disk driver receives a seek request and the controller is currently busy with an I/O transfer request on some other disk unit, the driver queues the request so that the controller will initiate the seek request before any pending I/O transfers when it has finished the current transfer.

When the device subsequently generates its interrupt at the hardware interrupt priority level, the interrupt dispatcher calls the appropriate interrupt service routine. An interrupt service routine simply saves the device control/status registers, requests a software interrupt at the driver's interrupt priority level, and returns to the interrupt dispatcher which is then free to scan for unit attentions. Because a disk controller cannot generate interrupts on any unit performing a seek until the current transfer completes, the interrupt dispatcher will also dispatch seek completion when dispatching a disk I/O transfer completion interrupt.

When the driver receives the completion interrupt, it prepares the I/O completion status for the requester, and requests a software interrupt. The driver is then free to process another request in its queue and, if the queue is not empty, the driver begins again. All I/O completion notification takes place outside the driver, minimizing the inter-request idle time. The I/O post routine notifies the process of I/O completion and releases or unlocks buffers.

COMPATIBILITY MODE OPERATING ENVIRONMENT

The processor can execute user mode PDP-11 instruction streams in the context of a process. The operating system supplements this feature by substituting its functionally equivalent system services for many of the RSX-11M operating system executive directives that user mode tasks may call. This enables the system to execute such non-privileged RSX-11M task images as:

- the PDP-11 MACRO assembler
- the PDP-11 COBOL compiler
- the PDP-11 BASIC-PLUS-2 compiler
- the RSX-11M program development and file management utilities, including the task builder, text editor, etc.

In addition, the operating system supports the RMS-11 and RMS-11K record management services procedures for compatibility mode programs. Program and data files can therefore be transported between VAX-11/780 and RSX systems.

The operating system also supports the RSX-11M Monitor Console Routine (MCR) commands, either typed directly on a terminal, or submitted as indirect command files.

User Programming Considerations

Any PDP-11 COBOL, PDP-11 BASIC-PLUS-2, FORTRAN IV/IAS-RSX, or PDP-11 MACRO program can be executed in compatibility mode, provided that it is first linked by the RSX-11M Version 3.1 task builder and that the resulting task image meets the following requirements:

- It must not execute PDP-11 privileged instructions.
- It must have been built for a mapped system.
- It must not depend on 32-word memory granularity.
- It must not use the privileges that enable it to map into the executive or I/O page.
- It must not use the PLAS (program logical address space) executive directives.
- It must not rely on environmental features of RSX-11M that VAX does not support, e.g., partitioning or significant events.
- It must not use DECnet.

The task can be privileged to issue directives other than memory management directives — direct volume access using the QIO request executive directive, for example. IAS or RSX-11D tasks that meet these requirements can also be executed. They must first be built with the RSX-11M Version 3.1 task builder. For programs that do not meet these requirements, VAX provides the program development utilities (for example, the MACRO assembler and the task builder) for modifying programs to execute in compatibility mode.

For all other RSX-11M executive directives, the native mode operating system executes a functionally equivalent system service. In most cases, the system service duplicates the function. For example:

- A checkpoint enable/disable directive is interpreted as the set swap mode system service.
- The send/receive directives are translated into mailbox write/read system services. Native mode and compatibility mode images can communicate using mailboxes.
- The event flag directives are for the most part identical. Native mode and compatibility mode images can communicate using common event flags, provided they are in the same group.
- A Logical Unit Number (LUN) assignment directive is interpreted as a channel assignment for the appropriate device.

In some cases the operating system cannot duplicate the function, but it does what it can to let a program continue. For example:

- A task image is allowed to declare a significant event, but the directive is ignored.
- A set priority directive is ignored, since the scheduling priority ranges are different. To run at a given priority, the image must be run in the context of a process given that priority.

For the most part, however, many RSX-11M and VAX program environment characteristics correspond. For example, tasks can hibernate, receive asynchronous system traps, and schedule wake requests. Synchronous system trap routines can be declared as condition handlers for trace traps, breakpoint traps, illegal instruction traps, memory protection violations, and odd address errors.

File System and Data Management

Both RSX-11M and VAX recognize User Identification Codes as a protection mechanism. UICs provide the default user file directory in RSX-11M systems, while, in VAX, a UIC is not necessarily associated with an account name or default directory name. UIC-based file protection, however, is much the same in both systems. That is, it is used in determining read, write, and delete privileges for system, owner, group, and world. UICs in VAX, however, can have twice the range of UICs in RSX-11M.

Tasks may use any of the RSX data management services including File Control Services (FCS), RMS-11, and RMS-11K. RMS-11 procedures are a compatible subset of the VAX RMS procedures. RMS-11K indexed accessing procedures are not available to native programming languages.

Both magnetic tape and Files-11 disk volumes can be transported between systems. VAX/VMS can read and write both Files-11 Level 1 (ODS-1) disk structures and the Level 2 disk structures (ODS-2). The Extend access protection field in ODS-1 is used for Execute access protection in ODS-2. While reading files stored on ODS-1 volumes, therefore, this protection field is ignored.

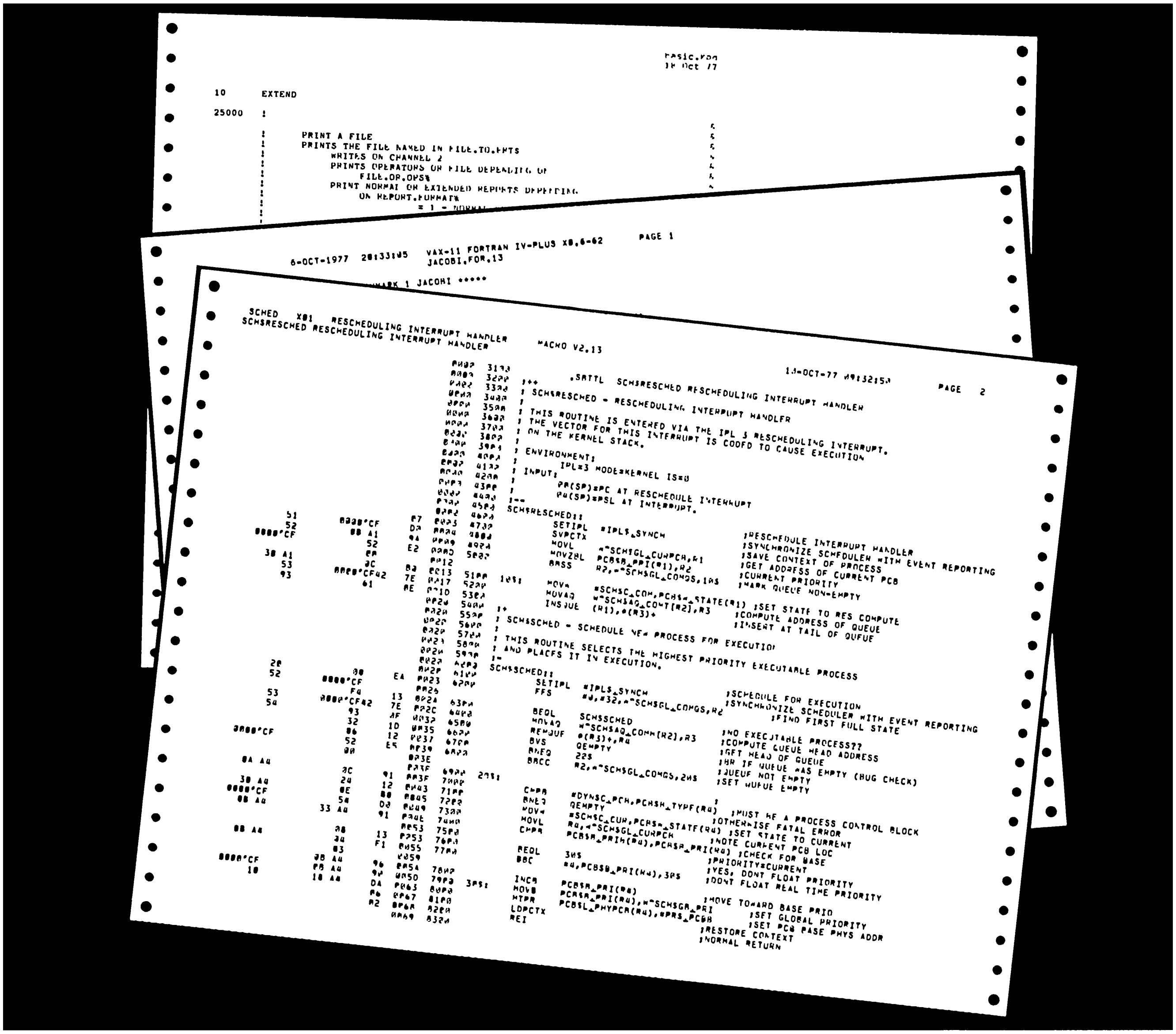
Command Languages

VAX/VMS users can select the MCR command interpreter, which allows them to execute a language (VAX/VMS MCR) that is similar to the RSX-11M MCR command language. Selecting the MCR command interpreter allows the VAX/VMS user to perform the following:

- Run RSX-11M images and VAX-11 images.
- Use RSX-11M components for RSX-11M program development, for example, MACRO-11 or the task builder.
- Use VAX/VMS components for native program development, for example, VAX-11 MACRO or the linker.
- Execute RSX-11M indirect command files. (The VAX/VMS user can use this facility to execute those files required for RSX-11M or RSX-11S system generation.)

VAX/VMS will associate the MCR command interpreter with a process, if "MCR" is the default command interpreter named in the user's authorization file entry or if the user specifies /CLI=MCR following "username" in the LOGIN statement (overriding the default).

7 The Languages



```
                                                   basic.ron
                                                   18 Oct 77

10      EXTEND

25000   !
        !       PRINT A FILE
        !       PRINTS THE FILE NAMED IN FILE.TO.PRTS
        !            WRITES ON CHANNEL 2
        !            PRINTS OPERATORS OR FILE DEPENDING ON
        !                 FILE.OR.OPS%
        !            PRINT NORMAL OR EXTENDED REPORTS DEPENDING
        !                 ON REPORT.FORMAT%
        !                              = 1 - NORMAL
```

```
6-OCT-1977  20:33:05   VAX-11 FORTRAN IV-PLUS X0.6-62      PAGE 1
                       JACOBI.FOR.13

                    MARK 1 JACOBI *****
```

```
SCHED    X01   RESCHEDULING INTERRUPT HANDLER
SCH$RESCHED RESCHEDULING INTERRUPT HANDLER      MACRO V2.13                13-OCT-77 09:32:50      PAGE  2

        .SBTTL  SCH$RESCHED RESCHEDULING INTERRUPT HANDLER
;++
; SCH$RESCHED - RESCHEDULING INTERRUPT HANDLER
;
; THIS ROUTINE IS ENTERED VIA THE IPL 3 RESCHEDULING INTERRUPT.
; THE VECTOR FOR THIS INTERRUPT IS CODED TO CAUSE EXECUTION
; ON THE KERNEL STACK.
;
; ENVIRONMENT:
;       IPL=3 MODE=KERNEL IS=0
; INPUT:
;
;       00(SP)=PC AT RESCHEDULE INTERRUPT
;       04(SP)=PSL AT INTERRUPT.
;--
SCH$RESCHED::
        SETIPL  #IPL$_SYNCH                     ;RESCHEDULE INTERRUPT HANDLER
        SVPCTX                                  ;SYNCHRONIZE SCHEDULER WITH EVENT REPORTING
        MOVL    W^SCH$GL_CURPCB,R1              ;SAVE CONTEXT OF PROCESS
        MOVZBL  PCB$B_PRI(R1),R2                ;GET ADDRESS OF CURRENT PCB
        BBSS    R2,W^SCH$GL_COMQS,10$           ;CURRENT PRIORITY
10$:    MOVW    #SCH$C_COM,PCB$W_STATE(R1)      ;MARK QUEUE NON-EMPTY
        MOVAQ   W^SCH$AQ_COMT[R2],R3            ;SET STATE TO RES COMPUTE
        INSQUE  (R1),@(R3)+                     ;COMPUTE ADDRESS OF QUEUE
                                                ;INSERT AT TAIL OF QUEUE
;+
; SCH$SCHED - SCHEDULE NEW PROCESS FOR EXECUTION
;
; THIS ROUTINE SELECTS THE HIGHEST PRIORITY EXECUTABLE PROCESS
; AND PLACES IT IN EXECUTION.
;-
SCH$SCHED::
        SETIPL  #IPL$_SYNCH                     ;SCHEDULE FOR EXECUTION
        FFS     #0,#32,W^SCH$GL_COMQS,R2        ;SYNCHRONIZE SCHEDULER WITH EVENT REPORTING
                                                ;FIND FIRST FULL STATE
        BEQL    SCH$SCHED                       ;NO EXECUTABLE PROCESS??
        MOVAQ   W^SCH$AQ_COMH[R2],R3            ;COMPUTE QUEUE HEAD ADDRESS
        REMQUE  @(R3)+,R4                       ;GET HEAD OF QUEUE
        BVS     QEMPTY                          ;BR IF QUEUE WAS EMPTY (BUG CHECK)
        BNEQ    20$                             ;QUEUE NOT EMPTY
        BBCC    R2,W^SCH$GL_COMQS,20$           ;SET QUEUE EMPTY
20$:
        CMPB    #DYN$C_PCB,PCB$B_TYPE(R4)       ;
        BNEQ    QEMPTY                          ;MUST BE A PROCESS CONTROL BLOCK
        MOVW    #SCH$C_CUR,PCB$W_STATE(R4)      ;OTHERWISE FATAL ERROR
        MOVL    R4,W^SCH$GL_CURPCB              ;SET STATE TO CURRENT
        CMPB    PCB$B_PRIB(R4),PCB$B_PRI(R4)    ;NOTE CURRENT PCB LOC
                                                ;CHECK FOR BASE
        BEQL    30$                             ;PRIORITY=CURRENT
        BBC     #4,PCB$B_PRI(R4),30$            ;YES, DONT FLOAT PRIORITY
30$:    INCB    PCB$B_PRI(R4)                   ;DONT FLOAT REAL TIME PRIORITY
        MOVB    PCB$B_PRI(R4),W^SCH$GB_PRI      ;MOVE TOWARD BASE PRIO
        MTPR    PCB$L_PHYPCB(R4),#PR$_PCBB      ;SET GLOBAL PRIORITY
        LDPCTX                                  ;SET PCB BASE PHYS ADDR
        REI                                     ;RESTORE CONTEXT
                                                ;NORMAL RETURN
```

VAX/VMS includes a complete program development environment. In addition to the native assembly language, it offers optional high-level programming languages commonly used in developing both scientific and commercial applications: FORTRAN, COBOL, BASIC, and RPG. It also offers the implementation language BLISS-32 for systems programming applications. VAX/VMS provides the tools necessary to write, assemble or compile, and link programs, as well as build libraries of source, object, and image modules.

Programmers can use the system for development while production is in progress. They can interact with the system on-line, execute command procedures, or submit command procedures as batch jobs. Novice programmers can learn the system quickly because the command language accepts standard defaults for invoking the editors, compilers, and linker. Experienced programmers will appreciate the flexibility and control each tool offers.

COMPONENTS

VAX/VMS provides two programming environments:

- the native programming environment
- the compatibility mode programming environment

The native programming environment consists of the language processors that produce native object code and the program development tools that support native program development. The VAX-11 MACRO assembler, VAX-11 FORTRAN IV-PLUS compiler, VAX-11 COBOL-74 compiler, and BLISS-32 compiler all produce VAX-11 native mode code.

The compatibility mode programming environment consists of the language processors that produce PDP-11 compatibility mode object code and the program development tools that support compatibility program development. The PDP-11 BASIC-PLUS-2/VAX compiler and the PDP-11 RPG II/VAX compiler produce compatibility mode code.

From the programmer's point of view, there is very little difference between native and compatibility program development. The difference between the native programming environment and the compatibility mode programming environment is that:

- Native programs can be built from procedures written in any native language, and native programs share a common run-time procedure library.
- Compatibility mode programs can be built from procedures written in a given compatibility mode language, and compatibility mode programs each have their individual object time libraries.

Furthermore, native and compatibility mode programs can read and write the same files, with the exceptions that native programs can read and write sequential and relative files only, while compatibility mode programs can read and write sequential, relative, and indexed files. Refer to the Data Management Services section for more information on file accessing.

The compatibility mode programming environment can be extended to provide an RSX-11M program development environment. The option includes RSX-11M language processors and tools that can be used to create programs to be executed in PDP-11 compatibility mode on the VAX-11/780 system, or to be executed on PDP-11 systems running the RSX-11M or RSX-11S operating systems, and in some cases, the IAS operating system.

VAX-11 MACRO

The VAX-11 MACRO assembler accepts one or more source modules written in MACRO assembly language and produces a relocatable object module and optional assembly listing. VAX-11 MACRO is similar to PDP-11 MACRO, but its instruction mnemonics correspond to the VAX-11/780 native instructions. VAX-11 MACRO is characterized by:

- relocatable object modules
- global symbols for linking separately assembled object programs
- global arithmetic, global assignment operator, global label operator and default global declarations
- user-defined macros with keyword arguments
- multiple macro libraries with fast access structure
- program sectioning directives
- conditional assembly directives
- assembly and listing control functions
- alphabetized, formatted symbol table listing
- default error listing on command output device
- a Cross Reference Table (CREF) symbol listing

Symbols and Symbol Definitions

Three types of symbols can be defined for use within MACRO source programs: permanent symbols, user-defined symbols and macro symbols. Permanent symbols consist of the VAX-11 instruction mnemonics and MACRO directives and do not have to be defined by the user. User-defined symbols are those used as labels or defined by direct assignment. Macro symbols are those symbols used as macro names.

MACRO maintains a symbol table for each type of symbol. The value of a symbol depends on its use in the program. To determine the value of a symbol in the operator field, the assembler searches the macro symbol table, user symbol table, and permanent symbol table in that order. To determine the value of the symbol used in the operand field, the assembler searches the user symbol table and the permanent symbol table in that order. These search orders allow redefinition of permanent ,symbol table entries as user-defined or macro symbols.

User-defined symbols are either internal to a source program module or global (externally available). An internal symbol definition is limited to the module in which it appears. Internal symbols are local definitions which are resolved by the assembler.

A global symbol can be defined in one source program module and referenced within another. Global symbols are preserved in the object module and are not resolved until the object modules are linked into an executable program. With some exceptions, all user-defined symbols are internal unless explicitly defined as being global.

Directives

A program statement can contain one of three different operators: a macro call, a VAX-11 instruction mnemonic, or an assembler directive. MACRO includes directives for:

- listing control
- function specification
- data storage
- radix and numeric usage declarations
- location counter control
- program termination
- program boundaries information
- program sectioning
- global symbol definition
- conditional assembly
- macro definition
- macro attributes
- macro message control

- repeat block definition
- macro libraries

Listing Control Directives

Several listing control directives are provided in MACRO to control the content, format, and pagination of all listing output generated during assembly. Facilities also exist for titling object modules and presenting other identification information in the listing output.

The listing control options can also be specified at assembly time through qualifiers in the command string issued to the MACRO assembler. The use of these qualifiers provides initial listing control options that may be overriden by the corresponding listing control directives in the source program.

Conditional Assembly Directives

Conditional assembly directives enable the programmer to include or exclude blocks of source code during the assembly process, based on the evaluation of stated condition tests within the body of the program. This capability allows several variations of a program to be generated from the same source module.

The user can define a conditional assembly block of code, and within that block, issue subconditional directives. Subconditional directives can indicate the conditional or unconditional assembly of an alternate or non-contiguous body of code within the conditional assembly block. Conditional assembly directives can be nested.

Macro Definitions and Repeat Blocks

In assembly language programming, it is often convenient and desirable to generate a recurring coding sequence by invoking a single statement within the program. In order to do this, the desired coding sequence is first established with dummy arguments as a macro definition. Once a macro has been defined, a single statement calling the macro by name with a list of real arguments (replacing the corresponding dummy arguments in the macro definition) generates the desired coding sequence or macro expansion. MACRO automatically creates unique symbols where a label is required in an expanded macro to avoid duplicate label specifications. Macros can be nested; that is, the definition of one macro can include a call to another.

An indefinite repeat block is a structure that is similar to a macro definition, except it has only one dummy argument. At each expansion of the indefinite repeat range, this dummy argument is replaced with successive elements of a specified real argument list. This type of macro definition does not require calling the macro by name, as required in the expansion of conventional macros. An indefinite repeat block can appear within or outside of another macro definition, indefinite repeat block, or repeat block.

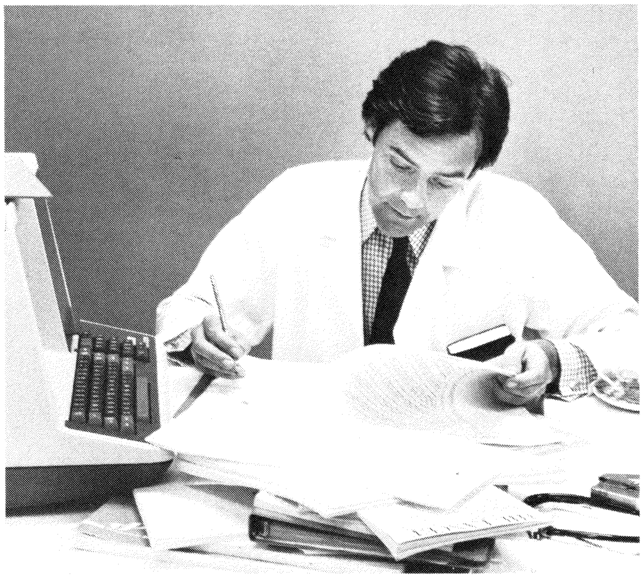
Macro Calls and Structured Macro Libraries

A program can call macros that are not defined in that program. A user can create libraries of macro definitions, and MACRO will look up definitions in one or more given library files when the calls are encountered in the program. Each library file contains an index of the macro definitions it contains to enable MACRO to find defintions quickly.

Program Sectioning

The MACRO program sectioning directives are used to declare names for program sections and to establish certain program section attributes. These program section attributes are used when the program is linked into an image.

The program sectioning directive allows the user to exercise complete control over the virtual memory allocation of a program, since any program attributes established through this directive are passed to the linker. For example, if a programmer is writing multi-user programs, the program sections containing only instructions can be declared separately from the sections containing only data. Furthermore, these program sections can be declared as read-only code, qualifying them for use as protected, reentrant programs.



VAX-11 FORTRAN IV-PLUS

VAX-11 FORTRAN IV-PLUS is an optional language processing system whose language specifications are based on the American National Standard FORTRAN X3.9-1966. VAX-11 FORTRAN IV-PLUS also includes many language features of the proposed American National Standard FORTRAN-77. The FORTRAN IV-PLUS compiler:

- produces highly optimized VAX-11 native object code
- makes use of the VAX-11 floating point and character string instructions
- produces sharable code

The VAX-11 FORTRAN IV-PLUS language is upward compatible with the PDP-11 FORTRAN IV and FORTRAN IV-PLUS languages. The VAX-11 FORTRAN IV-PLUS compiler supports the same enhancements to the language standard as PDP-11 FORTRAN IV and FORTRAN IV-PLUS, as well as providing additional enhancements. Table 7-1 lists most of the extensions to the ANS FORTRAN specification provided by the FORTRAN language compilers. Some characteristics of VAX-11 FORTRAN IV-PLUS are described in the following paragraphs.

Table 7-1
ANS FORTRAN Extensions

VAX-11 and PDP-11 FORTRAN IV-PLUS, PDP-11 FORTRAN IV

Array Dimensions	Arrays can have up to seven dimensions.
Array Subscripts	Any arithmetic expression can be used as an array subscript. If the value of the expression is not an integer, it is converted to integer format.
Character Literals	Character strings bounded by apostrophes can be used in place of Hollerith constants.
Mixed-mode Expressions	Mixed-mode expressions can contain any data type, including complex and byte.
End of Line Comments	Any FORTRAN statement can be followed, in the same line, by a comment that begins with an exclamation point.
Conditional Compilation of Debugging Statements	Statements that are included in a program for debugging purposes can be so designated by the letter D in column 1. Those statements are compiled only when the associated compiler command option is set. They are treated as comments otherwise.
General Expression DO and GO TO Parameters	General expressions are permitted for the initial value, increment, and limit parameters in the DO statement, and as the control parameter in the computed GO TO statement.
DO Increment Parameter	The value of the DO statement increment parameter can be negative.
Optional Statement Label List	The statement label list in an assigned GO TO is optional.
General Expressions in I/O Lists	General expressions are permitted in I/O lists of WRITE, TYPE, and PRINT statements.
Default FORMAT Widths	The programmer can specify input or output formatting by type and default width and precision values will be supplied.
List-Directed I/O Statements	The READ (u,*), WRITE (u,*), TYPE*, ACCEPT*, and PRINT* statements provide list directed, or "free format", I/O without requiring a FORMAT specification.
Additional I/O Statements	OPEN and CLOSE statements provide file control and attribute definition. ACCEPT, TYPE, and PRINT statements provide device-oriented I/O. ENCODE and DECODE statements provide memory-to-memory formatting. DEFINE FILE, READ (u'r), WRITE (u'r), and FIND (u'r) provide unformatted direct access I/O, which allows the FORTRAN programmer to read and write files written in any format.
End-of-file or Error Condition Transfer	The specifications END=n and ERR=n (where n is a statement label) can be included in any READ or WRITE statement to transfer control to the specified statement upon detection of an end-of-file or error condition. The ERR=n option is also permitted in the ENCODE and DECODE statements, allowing program control of data format errors.
Additional Data Type	The byte data type (keyword LOGICAL*1 or BYTE) is useful for storing small integer values as well as for storing and manipulating character information.
Logical Operations	The logical operators .AND., .OR., .NOT., .XOR., and logical .EQV. may be applied to integer data to perform bit masking and manipulation.
IMPLICIT declaration	The IMPLICIT statement has been added to redefine the implied data type of symbolic names.

VAX-11 and PDP-11 FORTRAN IV-PLUS

ENTRY statement	ENTRY statements can be used in SUBROUTINE and FUNCTION subprograms to define multiple entry points in a single program unit.
PARAMETER statement	PARAMETER statements can be used to give symbolic names to constants.
INCLUDE statement	The INCLUDE statement incorporates FORTRAN source text from a separate file into a FORTRAN program.
Generic function selection	Function selection by argument data type is provided for many FORTRAN library functions.
Array Dimension Bounds	Lower bounds as well as upper bounds of the array dimension can be specified in array declarators. The value of the lower bound dimension declarator can be negative, zero, or positive.
DO Loop Iteration Count	The terminal and increment parameters can be modified within a DO loop without affecting the iteration count. The number of times a DO loop is executed is determined at the initialization of the DO statement and is not re-evaluated during successive executions of the loop. Consequently, the number of times the loop is executed will not be affected by changing the variables used in the DO statement.

VAX-11 FORTRAN IV-PLUS

Long Symbolic Names	Symbolic names used to identify programs, subprograms, external functions and subroutines, COMMON blocks, variables, arrays, symbolic constants, and statement functions can be longer than the standard 6 characters. Symbolic names can be from 1 to 15 characters long and can include letters, digits, dollar sign, and underscore. The first character in the name must be a letter.

Table 7-1 (cont)
Language Extensions to ANS X3.9-1966 FORTRAN

Additional Data Type	The data type INTEGER*4 provides a sign plus 31 bits of precision. INTEGER*4 allows a greater range of values to be represented than INTEGER*2. Both data types can be used in the same program.
INTEGER Data Type Defaults	A compiler command specification allows all INTEGER and LOGICAL declarations without explicit length specifications to be considered as INTEGER*2 and LOGICAL*2, or INTEGER*4 and LOGICAL*4, respectively.
Additional I/O Statements	READ (u'r,fmt) and WRITE (u'r,fmt) provide input and output to direct access files.
DO Control Variable Data Types	The control variable of a DO statement can be a REAL or DOUBLE PRECISION variable, as well as an INTEGER*2 or INTEGER*4 variable. The initial, terminal, and increment parameters can be of any data type and are converted before use to the type of the control variable if necessary.
Hexadecimal Constants and Field Descriptor	Both octal and hexadecimal constants can be expressed in DATA statements. No conversion of the defined value (such as sign-extension) is performed. The Z field descriptor in FORMAT statements enables a program to read and write hexidecimal digits which are stored in an internal format in an I/O list element.
Additional Data Type	The data type CHARACTER permits manipulation of strings of ASCII characters expressed as constants, variables, arrays, substrings, symbolic names, or functions.
IF THEN ELSE Statements	The proposed FORTRAN-77 block-IF statements are provided: IF, ELSE IF, ELSE, and ENDIF. These structured programming statements provide more readable and reliable methods for expressing conditional statement execution.
Standard CALL Facility	Provides standard argument definitions for called procedures.

File Manipulation

The FORTRAN IV-PLUS OPEN and CLOSE statements extend the file manipulating characteristics of the FORTRAN language. The OPEN statement can contain specifications for file attributes that control file creation or subsequent processing. Attributes include: file organization (sequential, relative), method of access (sequential, direct), protection (read-only, read/write), record type (formatted, unformatted), record size, and file allocation or extension. The program can also specify whether the file can be shared, and whether the file is to be deleted or saved when closed. The OPEN statement can contain an ERR= keyword that specifies the statement to which control is transferred if an error is detected during OPEN.

Simplified I/O Formats

List-directed input and output statements provide a method for obtaining simple sequential formatted input or output without the need for FORMAT statements. On input, values are read, converted to internal format, and assigned to the elements of the I/O list. On output, values in the I/O list are converted to characters and written in a fixed format according to the data type of the value.

Character Data Type

A program can create fixed-length CHARACTER variables and arrays to store ASCII character strings. The VAX-11 FORTRAN IV-PLUS language provides a concatenation operator, substring notation, CHARACTER relational expressions, and CHARACTER-valued functions. CHARACTER constants, consisting of a string of printable ASCII characters enclosed in single quotes, can be assigned symbolic names using the PARAMETER statement.

Source Program Libraries

The INCLUDE statement provides a mechanism for writing modular, reliable, and maintainable programs by eliminating duplication of source code. A section of program text that is used by several program units, such as a COMMON block specification, can be created and maintained as a separate source file. All program units which reference the COMMON block then merely INCLUDE this common file. Any changes to the COMMON block will be reflected automatically in all program units after compilation.

Calling External Functions and Procedures

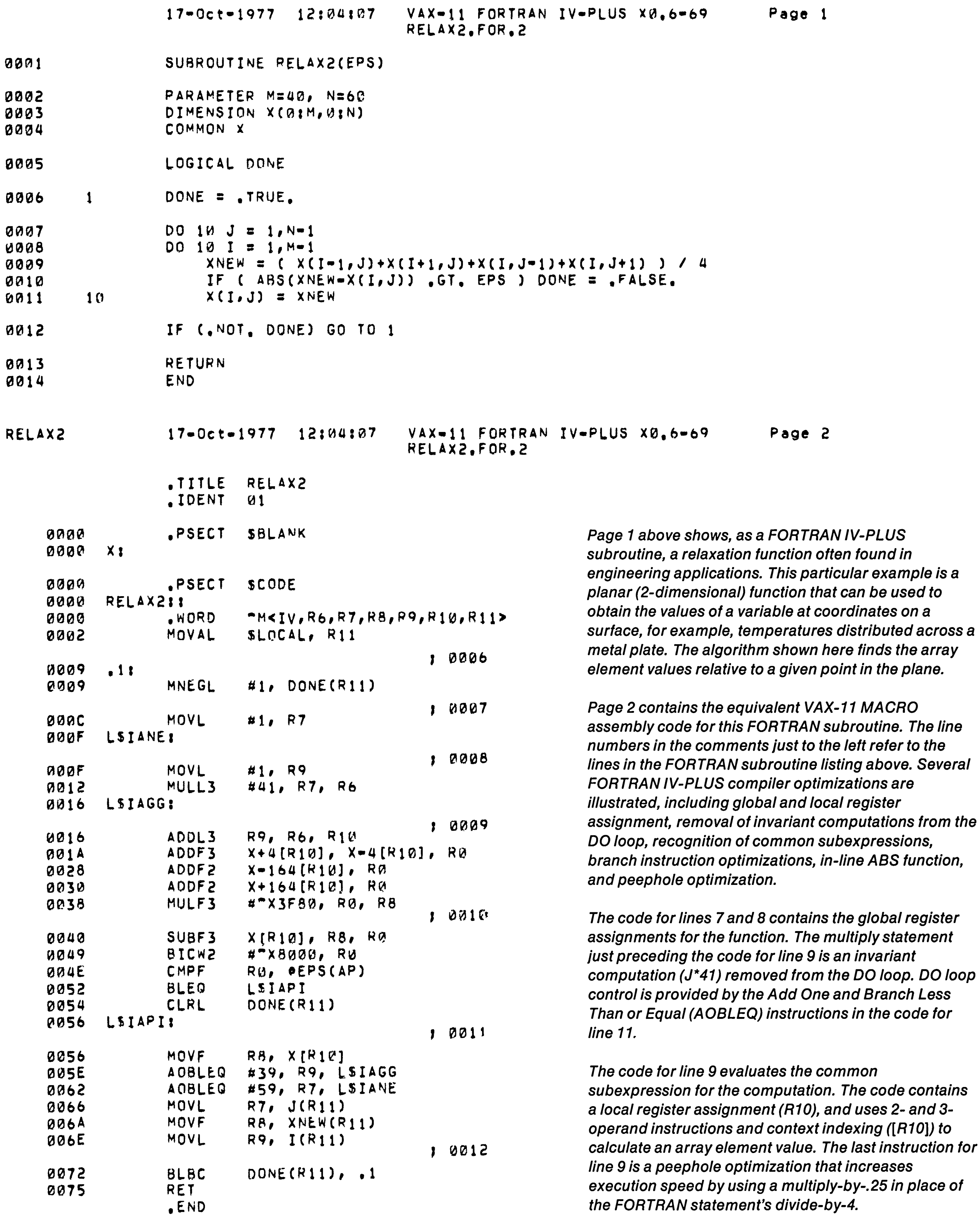
FORTRAN programs can call assembly language subroutines, and can call the system services and record management services using the standard VAX-11 procedure calling statements. Special operators exist for passing argument values directly, by reference, or by descriptor. A special operator also exists for obtaining the location of argument values.

Sharable Programs

The FORTRAN IV-PLUS language can be used to created sharable programs because the compiler can produce reentrant code. FORTRAN IV-PLUS subprograms can also be placed in sharable image libraries created by the linker, which can be made available to any program written in a native programming language.

Compiler Operation and Optimizations

The VAX-11 FORTRAN IV-PLUS compiler accepts source programs written in the FORTRAN language and produces an object file which must be linked prior to execution. The compiler generates VAX-11 native machine language code. Figure 7-1 is an illustration of FORTRAN IV-PLUS code and the equivalent VAX-11 assembly code.



```
                17-Oct-1977  12:04:07    VAX-11 FORTRAN IV-PLUS X0.6-69        Page 1
                                         RELAX2.FOR,2

0001            SUBROUTINE RELAX2(EPS)

0002            PARAMETER M=40, N=60
0003            DIMENSION X(0:M,0:N)
0004            COMMON X

0005            LOGICAL DONE

0006     1      DONE = .TRUE.

0007            DO 10 J = 1,N-1
0008            DO 10 I = 1,M-1
0009                XNEW = ( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1) ) / 4
0010                IF ( ABS(XNEW-X(I,J)) .GT. EPS ) DONE = .FALSE.
0011     10         X(I,J) = XNEW

0012            IF (.NOT. DONE) GO TO 1

0013            RETURN
0014            END


RELAX2          17-Oct-1977  12:04:07    VAX-11 FORTRAN IV-PLUS X0.6-69        Page 2
                                         RELAX2.FOR,2

                .TITLE  RELAX2
                .IDENT  01

    0000        .PSECT  $BLANK
    0000   X:

    0000        .PSECT  $CODE
    0000   RELAX2::
    0000        .WORD   ^M<IV,R6,R7,R8,R9,R10,R11>
    0002        MOVAL   $LOCAL, R11
                                                 ; 0006
    0009   .1:
    0009        MNEGL   #1, DONE(R11)
                                                 ; 0007
    000C        MOVL    #1, R7
    000F   L$IANE:
                                                 ; 0008
    000F        MOVL    #1, R9
    0012        MULL3   #41, R7, R6
    0016   L$IAGG:
                                                 ; 0009
    0016        ADDL3   R9, R6, R10
    001A        ADDF3   X+4[R10], X-4[R10], R0
    0028        ADDF2   X-164[R10], R0
    0030        ADDF2   X+164[R10], R0
    0038        MULF3   #^X3F80, R0, R8
                                                 ; 0010
    0040        SUBF3   X[R10], R8, R0
    0049        BICW2   #^X8000, R0
    004E        CMPF    R0, @EPS(AP)
    0052        BLEQ    L$IAPI
    0054        CLRL    DONE(R11)
    0056   L$IAPI:
                                                 ; 0011
    0056        MOVF    R8, X[R10]
    005E        AOBLEQ  #39, R9, L$IAGG
    0062        AOBLEQ  #59, R7, L$IANE
    0066        MOVL    R7, J(R11)
    006A        MOVF    R8, XNEW(R11)
    006E        MOVL    R9, I(R11)
                                                 ; 0012
    0072        BLBC    DONE(R11), .1
    0075        RET
                .END
```

Page 1 above shows, as a FORTRAN IV-PLUS subroutine, a relaxation function often found in engineering applications. This particular example is a planar (2-dimensional) function that can be used to obtain the values of a variable at coordinates on a surface, for example, temperatures distributed across a metal plate. The algorithm shown here finds the array element values relative to a given point in the plane.

Page 2 contains the equivalent VAX-11 MACRO assembly code for this FORTRAN subroutine. The line numbers in the comments just to the left refer to the lines in the FORTRAN subroutine listing above. Several FORTRAN IV-PLUS compiler optimizations are illustrated, including global and local register assignment, removal of invariant computations from the DO loop, recognition of common subexpressions, branch instruction optimizations, in-line ABS function, and peephole optimization.

*The code for lines 7 and 8 contains the global register assignments for the function. The multiply statement just preceding the code for line 9 is an invariant computation (J*41) removed from the DO loop. DO loop control is provided by the Add One and Branch Less Than or Equal (AOBLEQ) instructions in the code for line 11.*

The code for line 9 evaluates the common subexpression for the computation. The code contains a local register assignment (R10), and uses 2- and 3-operand instructions and context indexing ([R10]) to calculate an array element value. The last instruction for line 9 is a peephole optimization that increases execution speed by using a multiply-by-.25 in place of the FORTRAN statement's divide-by-4.

Figure 7-1
Sample VAX-11 FORTRAN IV-PLUS Program

During compilation, the FORTRAN IV-PLUS compiler performs many code optimizations. The optimizations are designed to produce an object program that executes in less time than an equivalent non-optimized program. The optimizations are also designed to reduce the size of the object program.

The FORTRAN IV-PLUS compiler performs the following optimizations:

- Constant folding. Integer constant expressions are evaluated at compile time.
- Compile-time constant conversion.
- Compile-time evaluation of constant subscript expressions in array calculations.
- Constant pooling. Only a single copy of a constant is allocated storage in the compiled program. Constants that can be used as immediate mode operands are not allocated storage. For example, logical, integer, and small floating point constants are generated as immediate mode or short literal operands wherever possible.
- Argument list merging. If two function or subroutine references have the same arguments, a single copy of the argument list is generated.
- Branch instruction optimizations for arithmetic or logical IF statements.
- Elimination of unreachable code. An optional warning message is issued to mark unreachable statements in the source program listing.
- Recognition and replacement of common subexpressions.
- Removal of invariant computations from DO loops.
- Local and global register assignment. Frequently referenced variables and expressions are retained in registers (locally and across DO loops) to reduce the number of memory references.
- Reordering expression evaluation to minimize the number of temporary registers required.
- Delaying negation/not to eliminate unary complement operations.
- Flow-Boolean optimizations.
- Jump/Branch instruction resolution. The Branch instruction is used wherever possible to eliminate unnecessary Jump instructions.
- Peephole optimization. The code is examined on an operation-by-operation basis to replace sequences of operations with shorter and faster equivalent operations.

DEBUGGING FACILITIES

VAX-11 FORTRAN IV-PLUS debugging facilities include diagnostic messages, conditional compilation flags, and access to the VAX/VMS DEBUG program. The DEBUG program lets the programmer set breakpoints and trace points, and examine and modify the contents of locations dynamically when executing the program.

DEBUG understands FORTRAN data type representations and syntax. It can examine and deposit locations using floating point representation, and it can reference FORTRAN symbols, statement labels, and line numbers symbolically. It can also reference arrays symbolically. For example:

```
EXAMINE A(I,J+3)
```

If the programmer is going to use the DEBUG program, the programmer can request the FORTRAN IV-PLUS compiler to disable optimizations that would remove unreferenced statement labels, FORMAT statement labels, and immediately referenced labels. This ensures that all statement labels are available to the debugger.

FORTRAN Symbolic Traceback

VAX-11 FORTRAN-IV PLUS supports the Symbolic Traceback Facility. This is a run-time facility that aids programmers in finding errors by describing the call sequences that occurred prior to the error. The Traceback facility is automatic and does not require that any special qualifiers be included with the FORTRAN or LINK commands (but it can be suppressed by specifying NOTRACE with the LINK command).

When an error condition is detected, the error message is diplayed by the run-time library indicating the nature of the error and the address at which the error occurred (user PC). This is followed by the traceback information, which is presented in inverse order to the calls. For each call frame, Traceback lists module name, routine name, source program line, and absolute and relative PC. Using this information, the programmer can usually locate the source of the error in a relatively short period of time. Figure 7-2 shows an example of a source program and traceback list. (Note that some of the entries in the list show relative and absolute PC but no corresponding values for module name and routine name; this indicates that the values refer to procedure calls internal to the run-time library.)

```
0001          I=1
0002          CONTINUE
0003          J=2
0004          CONTINUE
0005          K=3
0006          CALL SUB1
0007          CONTINUE
0008          END

0001          SUBROUTINE SUB1
0002          I=1
0003          J=2
0004          CALL SUB2
0005          END

0001          SUBROUTINE SUB2
0002          COMPLEX W
0003          COMPLEX Z

0004          DATA/(0.,0.)/

0005          Z = LOG(W)
0006          END
```

```
%MTH-F-LOGZERNEG, logarithm of zero or negative value
  user PC 00000449
%TRACE-F-TRACEBACK, symbolic stack dump follows
module name   routine name  line  relative PC    absolute PC

                                  0000074C       0000074C
                                  0000081C       0000081C
              SUB2           5    00000011       00000449
              SUB1           4    00000017       00000437
              T1$MAIN        6    0000001B       0000041B
```

Figure 7-2
FORTRAN Symbolic Traceback

VAX-11 COBOL-74

VAX-11 COBOL-74 is an optional language processing system that provides data processing for commercial applications. It conforms in language elements, representation, symbology, and coding format to ANS-COBOL Spec. X.3.23-1974 and is highly compatible with PDP-11 COBOL-74. The VAX-11 COBOL-74 compiler produces native mode code and thus takes advantage of the system's virtual address space, internal instruction set and data types.

VAX-11 COBOL-74 includes the following language elements.

- Level 2 Nucleus module
- Level 2 Table Handling module
- Level 2 Sequential I/O module
- Level 2 Relative I/O module
- Level 2 Indexed I/O module
- Level 2 Segmentation module
- Level 1 Library Module, with partial Level 2 REPLACING facility
- Level 1 Interprogram Communication Module
- Cross Reference Compilation Listing
- DISPLAY verb WITH NO ADVANCING clause
- Conditional variables — Data Division level 88
- Nested conditionals

File Organizations

Both the Sequential I/O and Relative I/O modules meet the full ANS-74 Level 2 standards and include the following COBOL verbs:

- OPEN EXTEND — Add records to a previously created sequential file without recopying the file.
- DYNAMIC ACCESS — Process a relative file both randomly and sequentially in the same program.
- START — Select positioning within a relative file for subsequent record retrieval.
- REWRITE/WRITE — Logically replace a record in a mass storage file and generate a new record.
- CLOSE LOCK — Protect a file from being opened by the current program a second time.
- LINAGE — Specify logical page format.

The Level 2 Indexed I/O module statements enable COBOL programs to use the multikey indexed record management services to process indexed files. Indexed files can be accessed sequentially, randomly, or dynamically using one or more keys to select records. The Environment Division RESERVE clause enables the user to specify the number of buffer areas for fast multikey processing.

Data Types

VAX-11 COBOL-74 supports a variety of data types, including:

- Numeric DISPLAY Data
  - Trailing overpunch sign
  - Leading overpunch sign
  - Trailing separate sign
  - Leading separate sign
  - Unsigned
  - Numeric-edited
- Numeric COMPUTATIONAL Data
  - 1-word fixed binary
  - 2-word fixed binary
  - 4-word fixed binary
- Alphanumeric DISPLAY Data
  - Alphanumeric
  - Alphabetic
  - Alphanumeric-edited
- Packed Decimal Data

String Manipulation

COBOL has the capability to manipulate data strings. It offers the INSPECT, STRING, and UNSTRING verbs that search for embedded character strings, with tally and replace. In addition, it is possible to join together or break out separate strings with various delimiters.

Interactive COBOL Programs

The Procedure Division ACCEPT and DISPLAY statements allow terminal-oriented interaction between a COBOL program and a user. This is useful, for example, in an order entry application.

The ACCEPT statement allows the terminal user to enter input lines which the COBOL program can interpret. The DISPLAY statement transfers a message to a specified device, normally the user's console. The statement can be modified by a special WITH NO ADVANCING clause (without automatic appending of carriage return and line feed) that allows the COBOL program to control the format of the message sent. This is especially useful when typing prompting messages on the console.

While the ACCEPT and DISPLAY statements are intended primarily for use with keyboard devices, COBOL allows the ACCEPT statement to accept input from a card reader or the batch input stream, and the DISPLAY statement to display data on a line printer.

SAMPLE COBOL-74 CODE

The sample VAX-11 COBOL-74 code illustrates some of the powerful language elements of COBOL-74. This coding illustrates an interactive COBOL program which will generate various types of reports depending upon user-specified options. The program operates upon an indexed information file via the dynamic access mode. Illustrated are three major COBOL-74 verbs: ACCEPT, DISPLAY and INSPECT.

In Figure 7-3, the program describes the file organization and the access mode. Also described are the primary and alternate keys used for accessing the file randomly.

```
00064
00065      INPUT-OUTPUT SECTION.
00066
00067      FILE-CONTROL.
00068
00069          SELECT CUSTOMER-FILE
00070          ASSIGN TO "CUSTOM.DAT"
00071          ORGANIZATION IS INDEXED
00072          ACCESS MODE IS DYNAMIC
00073          RECORD KEY IS CUST-CUST-NUMBER
00074          ALTERNATE RECORD KEY IS
00075          CUST-CUSTOMER-NAME
00076          FILE STATUS IS CUSTOMER-FILE-STATUS.
00077          SELECT STATEMENT-REPORT
00078          ASSIGN TO "STATEM.REP"
00079          FILE STATUS IS
00080          STATEMENT-REPORT-STATUS.
```

Figure 7-3
File Description

Figure 7-4 describes an options area in the working storage section. The programmer uses "88" level numbers allowing reference to coded information via symbolic names.

```
00174   01   OPTIONS-AREA.
00175             03   OPTIONS-AREA-CHAR        OCCURS 30                PIC      X(1).
00176
00177   01   A-COUNT                                          PIC      9(2).
00178
00179   01   OPTION-STORAGE.
00180             03   OPTION-ENTRY             OCCURS 8                 PIC      9(1).
00181   01   OPTION-VALUES REDEFINES OPTION-STORAGE.
00182   03        FILLER                                                 PIC      9(1).
00183             88  WANT-STATEMENTS                     VALUE 1   THRU 9.
00184   03        FILLER                                                 PIC      9(1).
00185             88  WANT-INVOICES                       VALUE 1   THRU 9.
00186   03        FILLER                                                 PIC      9(1).
00187             88  WANT-ALL-CATALOGS                   VALUE 1   THRU 9.
00188   03        FILLER                                                 PIC      9(1).
00189             88  WANT-SOME-CATALOGS                  VALUE 1   THRU 9.
00190   03        FILLER                                                 PIC      9(1).
00191             88  WANT-CREDIT-LIMIT-LETTERS           VALUE 1   THRU 9.
00192   03        FILLER                                                 PIC      9(1).
00193
00194   01   RECORD-COUNT                                     PIC       9(5) VALUE 0.
00195   01   STATEMENT-COUNT                                  PIC       9(5) VALUE 0.
00196   01   INVOICE-COUNT                                    PIC       9(5) VALUE 0.
00197   01   CREDIT-LIMIT-COUNT                               PIC       9(5) VALUE 0.
00198   01   CATALOG-COUNT                                    PIC       9(5) VALUE 0.
00199   01
```

Figure 7-4
Working Storage Area

In Figure 7-5, using the DISPLAY verb, the interactive COBOL program requests the user to specify an options selection. The user response is then transmitted to the program via the ACCEPT verb. The program uses the INSPECT verb to check that a valid response has been received.

```
00250     DISPLAY "  ENTER OPTIONS:".
00251     DISPLAY "  S  =  Print statements".
00252     DISPLAY "  I  =  Print invoices".
00253     DISPLAY "  CA  =  Mail all catalogs".
00254     DISPLAY "  CO  =  Mail selective catalogs".
00255     DISPLAY "  CL  =  Credit limit letters".
00256     ACCEPT OPTIONS-AREA.
00257     MOVE ALL ZERO TO OPTION-STORAGE.
00258     IF OPTIONS-AREA = SPACES
00259            DISPLAY "Discrepancy Report Only"
00260            GO TO CONFIRM-OPTIONS.
00261     MOVE 0 TO A-COUNT.
00262     INSPECT OPTIONS-AREA TALLYING
00263            OPTION-ENTRY  (1)  FOR ALL "S"
00264            OPTION-ENTRY  (2)  FOR ALL "I"
00265            OPTION-ENTRY  (3)  FOR ALL "CA"
00266            OPTION-ENTRY  (4)  FOR ALL "CO"
00267            OPTION-ENTRY  (5)  FOR ALL "CL".
00268
00269     IF OPTION-STORAGE = ALL ZERO
00270            DISPLAY "No options recognized"
00271            STOP RUN.
00272
00273     DISPLAY "Selected options:".
00274     IF WANT-STATEMENTS
00275            DISPLAY " Statements".
00276     IF WANT-INVOICES
00277     DISPLAY " Invoices".
00278     IF WANT-ALL-CATALOGS
```

Figure 7-5
Procedure Division Utilizing Interactive COBOL Verbs

Figure 7-6 illustrates the dynamic access method, i.e., shift from random to sequential access. The user moves zero to the primary record key, searches the file randomly, and commences sequential processing at the first non-zero number.

```
00302                OPEN INPUT CUSTOMER-FILE.
00303                MOVE "000000" TO CUST-CUST-NUMBER.
00304                START CUSTOMER-FILE
00305                       KEY IS > CUST-CUST-NUMBER.
00306                OPEN OUTPUT STATEMENT-REPORT.
00307
00308         ************************************************
00309
00310         MAINLINE SECTION.
00311         SBEGIN.
00312                READ CUSTOMER-FILE NEXT
00313                       AT END
00314                          GO TO END-PROCESS.
00315                ADD 1 TO RECORD-COUNT.
00316         *
00317         *                    Print statement if required.
00318         *
```

Figure 7-6
Random to Sequential Access

Source Library Facility

With COBOL, the user has a full ANS-74 Level 1 Library facility, plus high-level extensions. All frequently used data descriptions and program text sections can be stored in library files available to all programs. These files can then be copied into source programs to reduce program preparation time and eliminate a common source of errors.

CALL Facility

VAX-11 COBOL-74 supports the CALL statement, allowing COBOL programs to invoke separately compiled subprograms and to pass arguments between them. These subprograms may be written in COBOL or in another VAX-11 supported language and may include system service routines written in MACRO-11. To facilitate calls between programs written in different languages, VAX-11 COBOL-74 provides an extended call facility which allows arguments to be passed by value, reference, or descriptor; this is in contrast to standard COBOL in which arguments are passed by reference only. Figure 7-7 shows a sample program which utilizes all three types of argument passing mechanisms.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALLTST2.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.  VAX-11/780.
OBJECT-COMPUTER.  VAX-11/780.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  TIMLEN       PIC  9(4) USAGE IS COMP VALUE IS 0.
01  D-TIMLEN     PIC  9(4) VALUE IS 9999.
01  TIMBUF       PIC  X(24) VALUE IS SPACES.
01  DUMMY        PIC  9(5) USAGE IS COMP VALUE IS 0.
01  RETURN-VALUE PIC 9(9) USAGE IS COMP
                 VALUE IS 999999999.
01  D-RETURN-VALUE PIC 9(9) VALUE IS 999999999.
PROCEDURE DIVISION.
PO.
    DISPLAY "CALL SYS$ASCTIM".
    CALL "SYS$ASCTIM" USING BY REFERENCE TIMLEN
                            BY DESCRIPTOR TIMBUF
                            BY VALUE DUMMY
                            BY VALUE DUMMY
                            GIVING RETURN VALUE.
    DISPLAY "DATE/TIME = " TIMBUF.
    MOVE TIMLEN TO D-TIMLEN.
    DISPLAY "LENGTH OF RETURNED = " D-TIMLEN.
    MOVE RETURN-VALUE TO D-RETURN-VALUE.
    DISPLAY "RETURN-VALUE = " D-RETURN-VALUE.
    STOP RUN.
```

Figure 7-7
System Services Call

In this program, the system service routine $ASCTIM is called, which converts binary time to an ASCII string representation. In this example, the buffer length as specified in "timbuf" plus the value of the item "dummy" determine the type of information which the service routine will return to the COBOL program (e.g., specifying a length of 24 plus values of 0 in the following two arguments will cause both current date and time to be returned; if a length of 11 had been specified, then only the date would be returned).

External Subprograms

The CALL statement enables a COBOL program to execute routines that are external to the object module in which the CALL statement appears. The COBOL-74 compiler produces an object module from a single source module supplied as input. The object module file can be linked with other COBOL object modules, or with object modules created by other VAX-11/780 native mode compilers to produce an executable image.

Debugging COBOL Programs

To make program debugging easier, the COBOL compiler produces source language listings with embedded diagnostics. Fully descriptive diagnostic messages are listed at the point of error. Over 500 different error conditions are checked, varying from simple warnings to fatal error detections.

When the compiler detects an error in the source program, the compiler attempts to recover from an error and to continue to compile the program. The kind of error message, whether informational, warning, or fatal, indicates the likelihood that the assumption made to recover from the error will produce an object program that will run as the programmer intended. The compiler will not produce object code if fatal errors are detected at compile time.

Debugging large source programs is made still easier by the use of the optional allocation maps for the Data and Procedure Divisions, external subprogram references, and object library references. The compiler can also produce a cross-reference listing.

When a fatal error occurs at run time, an error message identifying the cause of the error is displayed to the user. In addition, VAX-11 COBOL-74, like VAX-11 FORTRAN IV-PLUS, supports the Symbolic Traceback Facility, a run-time facility which describes the call sequences which occurred prior to the error. For each call frame, traceback displays the module name, routine name, source line number, and program-counter information. This information identifies in source language terms the module, routine, and source line in which the fatal error occurred.

Figure 7-8 illustrates the printing of an error message and the subsequent traceback for a COBOL module in which a subscript violation occurs at run time. The "module name" and "routine name" fields identify the entry point, SUBERRTST, into the COBOL module. The subscript violation occurs on line number 15 of the source module. The "relative PC" field specifies that the subscript violation correspondingly occurs at "3C" hexadecimal bytes into the object code relative to the entry point SUBERRTST. The "absolute PC" field also specifies that the violation occurs at absolute location "3074" in the executable image containing SUBERRTST. Thus, the issuance of a specific, English-like error message coupled with the traceback facility offers the user a powerful debugging tool in identifying programming errors.

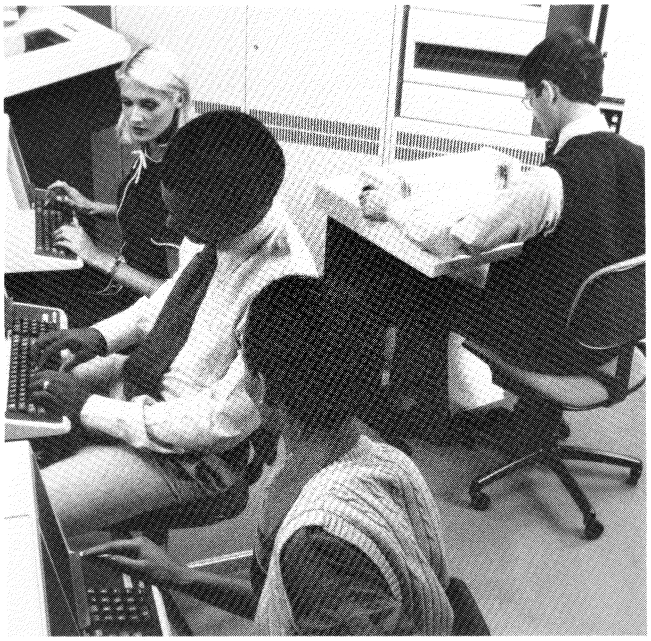
```
%C74-F-SUBOUTRAN, subscript out of range
%TRACE-F-TRACEBACK, symbolic stack dump follow
module name  routine name  line  relative PC  absolute PC

SUBERRTST    SUBERRTST     15    0000003C     00003074
```

Figure 7-8
Example of Traceback Facility

Source Program Formats

The COBOL compiler accepts source programs that are coded using either the conventional 80-column card reference format or a shorter, easy-to-enter terminal format. Terminal format is designed for use with the interactive text editors. It eliminates the line-number and identification fields and allows the user to enter horizontal tab characters and short lines.

The RFRMT (Reformat) utility program reads COBOL source programs that were coded using terminal format and converts the source statements to the 80-column source line format accepted by other COBOL compilers thoughout the industry. The programmer can enter source programs in the simpler terminal format and then, if compatibility is ever required with other systems, run the RFRMT utility to convert the source to conventional format.



PDP-11 BASIC-PLUS-2

PDP-11 BASIC-PLUS-2/VAX is an optional language processing system that includes a compiler and an object time system. PDP-11 BASIC-PLUS-2/VAX is the same BASIC-PLUS-2 language processing system available as an optional language processor for the RSTS/E, RSX-11M and IAS operating systems. The PDP-11 BASIC-PLUS-2/VAX compiler produces code that executes in PDP-11 compatibility mode.

BASIC-PLUS-2 is an extended BASIC language which features programming facilities not found in most BASIC languages. These include:

- program formatting and commenting facilities
- long variable names
- virtual arrays
- PRINT USING statement
- COMMON statement
- a subprogram CALL statement
- extended debugging facilities

The compiler accepts a source program written in the BASIC-PLUS-2 language. The programmer has several ways to use the BASIC-PLUS-2 compiler. The programmer can request the compiler to:

- load in a source program for editing
- compile a source program, produce an executable load module, and execute it ("load and go")
- compile a source program and produce an object module which can be linked with previously compiled object modules

The object time system is a collection of library modules used during program execution. The library routines include math and floating point functions, input/output operations, error handling, and dynamic string storage functions. Since the OTS is a library, the linker can select only those functions needed at run-time to be included in a program. Unnecessary routines are omitted from the program and memory usage is reduced.

Program Format

The BASIC source program unit is a line. In its simplest form, it consists of a line number, a keyword and statement, and a line terminator. In BASIC-PLUS-2, one or several spaces or tabs can be used to separate line numbers, keywords, and variable names. Line number determines the order in which the program is processed; the programmer can write BASIC-PLUS-2 program lines in any order.

BASIC-PLUS-2 programs can be one or several lines long. The programmer can place one statement on each line, place several statements on any one line, or spread one statement over several lines. Program comments can be placed anywhere within a line using the REM (Remark) statement or using comment field delimiters. These facilities enable the programmer to freely format a program to make it more readable.

Long Variable and Function Names

Most BASIC languages limit the length of a variable or user-defined function name to one character. BASIC-PLUS-2 recognizes variable names and function names as long as thirty characters. The programmer can fully identify variables and functions.

Dynamic String Handling

The BASIC-PLUS-2 language enables the programmer to manipulate strings of alphanumeric characters easily. The BASIC-PLUS-2 relational operators enable programmers to concatenate and compare strings; string operators enable the programmer to convert strings and numerics; and string functions add the ability to analyze the composition of strings. The BASIC-PLUS-2 language includes string functions that:

- create a string of a given length
- search for the position of a set of characters within a string
- insert spaces within a string
- trim trailing blanks from a string
- determine the length of a string

Unlike many BASIC languages, BASIC-PLUS-2 imposes no limit on the size of string scalars or string elements of arrays manipulated in memory other than the amount of memory available to the program.

Virtual Arrays

Virtual arrays are random access disk-resident files. A program can create and access virtual arrays in the same way memory-resident arrays are accessed: using element names. Explicit read/write programming is not required. The last element in the array can be accessed as quickly as the first. Because the arrays are stored on disk, however, the programmer can manipulate large amounts of data without affecting program size.

PRINT USING Output Formats

The PRINT USING statement allows the programmer to control the appearance and location of data on an output line to create complex lists, tables, reports, and forms. In addition to numeric field definitions that allow the programmer to generate floating dollar sign, aligned decimal point, trailing minus, asterisk fill, and exponential format fields, BASIC-PLUS-2 provides string field definitions which allow the programmer to generate left-justified, right-justified, centered, and extended string fields.

Subprograms and the CALL Statement

The BASIC-PLUS-2 CALL statement enables a program to access external subprograms. A programmer can, therefore, write a program in several modular segments, each of which can be compiled separately to speed program development. BASIC-PLUS-2 provides a complete traceback on errors occurring in subroutines.

COMMON Statement

The COMMON statement enables a program to pass data to another program or subprogram. The BASIC-PLUS-2 COMMON statement format is similar to FORTRAN COMMON. Strings passed in COMMON are fixed length, which reduces string handling overhead.

Debugging Statements

BASIC-PLUS-2 provides an interactive debugging mode similar to the "immediate mode" facilities found in most BASIC interpreters. During program development, the programmer can use the compiler to create, save, edit, and test the source program. The compiler checks syntax immediately on input from a terminal so that many errors can be found prior to compilation. The debugging statements can be used when executing and testing the program. The BREAK, LET, PRINT, UNBREAK, CONTINUE, STEP, and STOP statements enable the programmer to control and observe program execution interactively.

To set breakpoints, the programmer uses the BREAK command just prior to running the program, or while it is stopped. As many as 10 breakpoints can be set during the course of program execution. On reaching a breakpoint, the program halts to allow the programmer to examine or modify variables or set other breakpoints.

To examine variables while a program is stopped, the programmer uses the PRINT statement. The LET statement allows the programmer to modify the value stored in the variable.

Typing the CONTINUE command resumes execution until the next breakpoint is reached. Before typing CONTINUE, the programmer can issue an UNBREAK command to selectively disable one or all of the breakpoints set, and execution continues until a STOP statement is encountered in the program or the program completes.

When a program halts because a STOP statement is included in the program or because a BREAK command was issued interactively, the programmer can type the STEP command on the terminal to let program execution continue on a line-by-line basis. Typing a STOP command in interactive debugging mode terminates program execution, just as if an END statement was encountered in the program.

RPG II

PDP-11 RPG II/VAX is an optional language processing system that includes a compiler and an object time library system. It is a language particularly suited to producing printed output. The PDP-11 RPG II/VAX compiler produces code that executes in PDP-11 compatibility mode.

Language Elements

RPG II programming involves coding an ordered set of source specifications. Specifications have seven possible formats:

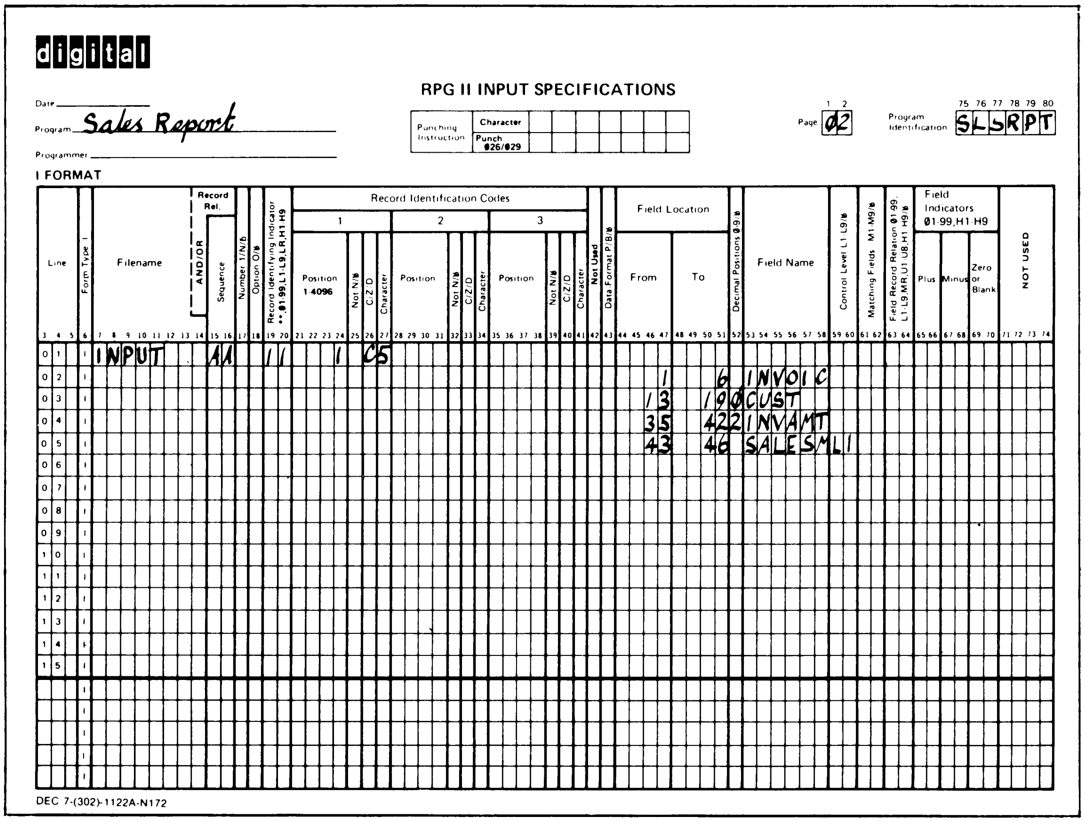
- the Control specifications (H Format), which supply information pertaining to the compilation as a whole
- the File Description specifications (F Format), which describe files to be used by the program
- the Extension specifications (E Format), which describe the tables and arrays to be used by the program and provide for additional file information
- the Line Counter specifications (L Format), which give special information about print output
- the Input specifications (I Format), which describe input records and fields
- the Calculation specifications (C Format), which describe the operations to be performed on previously specified data and define the data fields which are not previously defined
- the Output specifications (O Format), which describe the format of output records and the types of data fields

Figure 7-9 shows several examples of specification statements.

Special Features

RPG II provides a set of 31 instructions. These include standard arithmetic instructions, move instructions, compare and GOTO for looping and procedure branching, and instructions for defining and transferring to subroutines. In addition, RPG II provides a number of special features. These include:

- Table and Array Handling Facilities. These include special lookup operations allowing matches on low, high, or equal values; ability to load tables at compile time or as input at execution time, and use of the special arithmetic operation, XFOOT, to sum elements of an array.
- Ability to specify up to nine matching fields to control multifile processing.
- Ability to access records out of normal sequence via the FORCE and READ operations.
- Control of random record access via the CHAIN operation.
- The ability to specify multiple edit operations via a simple edit code.



digital

RPG II INPUT SPECIFICATIONS

Date

Program Sales Report

Programmer

Punching Instruction	Character								
	Punch Ø26/Ø29								

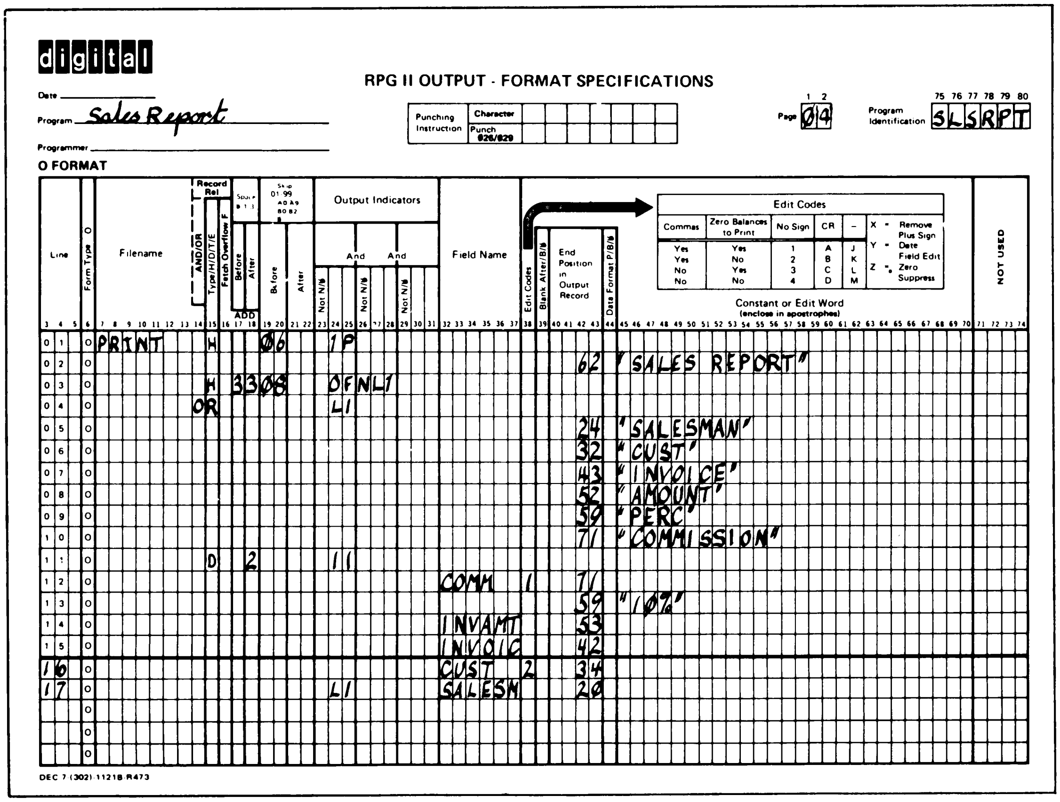
Page (1 2) Ø2

Program Identification (75 76 77 78 79 80) SLSRPT

I FORMAT

Line (3-5)	Form Type (6)	Filename (7-14)	AND/OR (14-16)	Record Rel. Sequence (15-16)	Number 1/N/b (17)	Option O/b (18)	Record Identifying Indicator **,Ø1-99,L1-L9,LR,H1-H9 (19-20)	Record Identification Codes 1 Position 1-4096 (21-24)	Not N/b (25)	C/Z/D (26)	Character (27)	2 Position (28-31)	Not N/b (32)	C/Z/D (33)	Character (34)	3 Position (35-38)	Not N/b (39)	C/Z/D (40)	Character (41)	Not Used (42)	Data Format P/B/b (43)	Field Location From (44-47)	To (48-51)	Decimal Positions Ø-9/b (52)	Field Name (53-58)	Control Level L1-L9/b (59-60)	Matching Fields M1-M9/b (61-62)	Field Record Relation Ø1-99 L1-L9,MR,U1-U8,H1-H9/b (63-64)	Field Indicators Ø1-99,H1-H9 Plus (65-66)	Minus (67-68)	Zero or Blank (69-70)	NOT USED (71-74)
0 1	I	INPUT		AA			11	1		C	5																					
0 2	I																					1	6		INVOIC							
0 3	I																					13	19	Ø	CUST							
0 4	I																					35	42	2	INVAMT							
0 5	I																					43	46		SALESM	L1						
0 6	I																															
0 7	I																															
0 8	I																															
0 9	I																															
1 0	I																															
1 1	I																															
1 2	I																															
1 3	I																															
1 4	I																															
1 5	I																															
	I																															
	I																															
	I																															
	I																															
	I																															
	I																															

DEC 7-(302)-1122A-N172



digital

RPG II OUTPUT - FORMAT SPECIFICATIONS

Date

Program Sales Report

Programmer

Punching Instruction	Character								
	Punch Ø26/Ø29								

Page (1 2) Ø4

Program Identification (75 76 77 78 79 80) SLSRPT

O FORMAT

Edit Codes

Commas	Zero Balances to Print	No Sign	CR	-
Yes	Yes	1	A	J
Yes	No	2	B	K
No	Yes	3	C	L
No	No	4	D	M

X = Remove Plus Sign
Y = Date Field Edit
Z = Zero Suppress

Line (3-5)	Form Type (6)	Filename (7-14)	Record Rel AND/OR (14-16)	Type H/D/T/E (15)	Fetch Overflow F (16)	Space Before (17)	Space After (18)	Skip Ø1-99 AØ-A9 BØ-B2 Before (19-20)	Skip After (21-22)	Output Indicators Not N/b (23)	(24-25)	And Not N/b (26)	(27-28)	And Not N/b (29)	(30-31)	Field Name (32-37)	Edit Codes (38)	Blank After B/b (39)	End Position in Output Record (40-43)	Data Format P/B/b (44)	Constant or Edit Word (enclose in apostrophes) (45-70)	NOT USED (71-74)
0 1	O	PRINT		H				Ø6			1P											
0 2	O																		62		'SALES REPORT'	
0 3	O			H		3	3	Ø8			OF	N	L1									
0 4	O		OR								L1											
0 5	O																		24		'SALESMAN'	
0 6	O																		32		'CUST'	
0 7	O																		43		'INVOICE'	
0 8	O																		52		'AMOUNT'	
0 9	O																		59		'PERC'	
1 0	O																		71		'COMMISSION'	
1 1	O			D			2				11											
1 2	O															COMM	1		71			
1 3	O																		59		'1Ø%'	
1 4	O															INVAMT			53			
1 5	O															INVOIC			42			
16	O															CUST	2		34			
17	O										L1					SALESM			2Ø			
	O																					
	O																					
	O																					

DEC 7-(302)-1121B-R473

Figure 7-9
Sample RPG-II Specifications

- "Look Ahead" feature, allowing fields on the next record awaiting processing to be accessed while the current record is being processed.
- Ability to set and test bits of a one-byte binary field.

File Organizations and Access Methods

RPG II supports the Sequential, Direct, and Indexed file organizations. Records may be accessed consecutively, sequentially by key, sequentially within limits, randomly by key, randomly by relative record number, and randomly using ADDress ROUTing files.

ADDROUT files are produced by the SORT utility program. They contain relative address pointers which may be used as indices to access the data in the original file. Since more than one ADDROUT file may be constructed vis-a-vis a single data file, the ADDROUT construct affords a method for accessing that file using several different sequences.

RPG II Utilities

In order to facilitate the coding of RPG II source specifications on the terminal, PDP-11 RPG II/VAX provides a special editor (RPGESP) which displays form templates, column numbers, and program statements. RPGESP allows the programmer to "fill in" columns on the display screen as if he were filling in a coding sheet. The programmer selects the appropriate source specification form by specifying the letter format designator (e.g., H for control, I for input, E for extensions, etc.) in the column following the sequence number. RPGESP performs syntax checking as information is entered at the terminal and displays a message if an error has been committed.

The RPGPCV utility program converts IBM System/3 RPG II source programs into a form which can be processed by the PDP-11 RPG II/VAX compiler. RPGPCV processes an input file of IBM System/3 RPG II statements. The utility automatically changes those fields containing language features supported by the PDP-11 RPG II/VAX compiler with or without a warning message. If an unsupported feature is encountered, then the field is left unchanged and a "fatal error" message is issued. If no fatal error has been detected, then RPGPCV creates three files — the converted source file, the original input source file as backup, and a listing file containing source statements and error messages. In the fatal error case, the programmer may opt to create the three files but the default will be to produce the listing file only.

The RPGDCV utility program converts an IBM System/3 standard labeled or unlabeled magnetic tape, fixed length, EBCDIC format data file with or without packed decimal fields into ASCII format. The input tape must have a density of either 800 or 1600 bpi. The converted output is stored on disk for further processing.

Debugging

RPG II features a special operation code, DEBUG, which will cause the system to generate information helpful in solving program errors. Information listed includes:

- data contained in a specific field
- a list of indicators which are on at the time indicated

The DEBUG operation is designed to be used at critical points in the RPG II source program and is coded on the H (Control) and C (Calculation) specification forms. The information requested will appear on an output file specified on the C form.

BLISS-32

BLISS-32 is a high-level systems implementation language for VAX-11. It is specifically designed for building software such as compilers, real-time processors, and operating systems modules. It contains many of the features associated with high level languages (e.g., DO loops, IF-THEN-ELSE statements, automatic stack, and mechanisms for defining and calling routines), but also provides the flexibility and access to hardware which one would expect from an assembly language. The BLISS-32 compiler runs in native mode under the VAX/VMS operating system.

Features of BLISS-32

BLISS-32 has several characterisics which set it apart from other higher-level languages:

- Data — All BLISS-32 calculations are performed on 32-bit values; the interpretation of any data item is provided by the instruction operator. BLISS-32 has the power to access and assign values to "variable bit" fields (i.e., contiguous fields of from 1 to 32 bits located anywhere in the virtual address space).
- Value Assignments — All names in BLISS-32 represent addresses. Contents of storage locations are accessed by means of a *fetch operator* (.). Hence, the expression X =.Y+3 is interpreted as adding 3 to the contents of location Y, then assigning the result to the storage location beginning at X.
- Expressions — BLISS-32 permits construction of complex expressions in which several different kinds of operations can be performed in a single program statement. For example, the expression 2*(B=.C+1) calculates 2*(.C+1) and simultaneously assigns the value of .C+1 to B.
- Structures — BLISS-32 defines data structures such as vectors, blocks, bitvectors, and blockvectors. In addition, the programmer can define arbitrary data structures specifically designed for a given application.

Other BLISS-32 features include:

- IF, CASE, SELECT, SELECTONE, and IF-THEN-ELSE sequences — providing for sequencing of instructions based upon evaluation of expressions at run-time.
- DO, WHILE, and UNTIL — providing for looping until a particular condition is satisfied.
- GLOBAL and EXTERNAL data declarations — allowing code to be shared between several modules
- LOCAL, STACKLOCAL, and REGISTER declarations — allowing dynamic stack-like allocation using either the execution stack of the VAX-11 or the VAX-11 general registers.
- REQUIRE and LIBRARY declarations — allowing outside files to be included in the module at compile time.
- LEXICAL functions — allowing a variety of compile-time operations such as concatenation of strings, construction of names, testing properties of macro parameters, testing compiler switch options, generating compiler diagnostic messages, and controlling macro expansion.

BLISS-32 also provides a number of features specifically designed to utilize or interact with the VAX-11 machine or VAX-VMS. These include the following:

- LINKAGES declarations allow the programmer to make full use of the VAX-11 call facilities; in particular, he may specify either the CALLS/CALLG/RET or JSB/BSB/RSB call and return sequences, pass parameters in general registers or parameter blocks, and control the use of registers by a routine or across a set of routines.
- PSECT declarations provide information to the linker regarding storage requirements for various sections of a program. For example, a particular data segment may be designated as READ or NOREAD, SHARE or NOSHARE, LOCAL or GLOBAL, and so on.
- BUILTIN declarations allow use of VAX-11 machine-specific functions for access to VAX-11 features not otherwise provided in the BLISS-32 language. The compilation of a machine-specific function results in the generation of internal code, often a single instruction, rather than a call to an external routine. Machine specific functions generally have the same name as their corresponding VAX-11 instructions (e.g., ADAWI, BISPSW, CRC, HALT, INDEX, MTPR, PROBER, REMQUE, MOVP, etc). Over 50 such functions are provided. (The complete list is shown in Table 7-2.)

Table 7-2
VAX-11 Machine-Specific Functions

Processor Register Operations	
MTPR	Move to a Processor Register
MFPR	Move from a Processor Register
Parameter Validation Operations	
PROBER	Probe Read accessibility
PROBEW	Probe Write accessibility
Program Status Operations	
MOVPSL	Move from PSL
BISPSW	Bit set PSW
BICPSW	Bit clear PSW
Queue Operations	
INSQUE	Insert entry in Queue
REMQUE	Remove entry from Queue
Bit Operations	
TESTBITSS	Test for Bit Set, then Set bit
TESTBITSC	Test for Bit Set, then Clear bit
TESTBITCS	Test for Bit Clear, then Set bit
TESTBITCC	Test for Bit Clear, then Clear bit
TESTBITSSI	Test for Bit Set, then Set bit Interlocked
TESTBITCCI	Test for Bit Clear, then Clear bit Interlocked
FFS	Find First Set bit
FFC	Find First Clear bit
Extended Arithmetic Operations	
ASHQ	Arithmetic Shift Quad
EDIV	Extended Divide
EMUL	Extended Multiply
INDEX	Index (Subscript) Calculation
CRC	Cyclic Redundancy Calculation
Floating Point Conversion Operations	
CVTLF	Convert Long to Floating
CVTLD	Convert Long to Double
CVTFL	Convert Floating to Long
CVTDL	Convert Double to Long
CVTFD	Convert Floating to Double
CVTDF	Convert Double to Floating
CVTRDL	Convert Rounded Double to Long
CVTRFL	Convert Rounded Floating to Long
CMPF	Compare Floating
CMPD	Compare Double
String Operations	
MOVTUC	Move Translated Until Character
SCANC	Scan Characters
SPANC	Span Characters
Decimal String Operations	
MOVP	Move Packed
CMPP	Compare Packed
CVTLP	Convert Long to Packed
CVTPL	Convert Packed to Long
CVTPT	Convert Packed to Trailing Numeric
CVTTP	Convert Trailing Numeric to Packed
CVTPS	Convert Packed to Leading Separate Numeric
CVTSP	Convert Leading Separate Numeric to Packed
EDITPC	Edit Packed to Character
Miscellaneous Operations	
HALT	Halt Processor
ROT	Rotate
ADAWI	Add Aligned Word Interlocked
BPT	Breakpoint
CHMx	Change Mode
CALLG	Call with General Argument List
NOP	No Operation

The BLISS-32 Compiler

The BLISS-32 compiler performs a number of optimizations. These include: common subexpression elimination, removing loop invariants, constant folding, block register allocation, peephole replacement, test instruction elimination, jump vs. branch instruction, branch chaining, and cross-jumping.

The BLISS-32 compiler optionally produces source text and generated code in a format closely resembling a VAX-11 assembly listing. Other options allow the programmer to control the degree of optimization, suppress production of object code, determine types and formats of output listings, generate traceback information, and specify the types of information to be listed at the terminal.

Library and Require Files

BLISS-32 provides two methods for including commonly used text in BLISS-32 programs at compile time. These involve use of either *Require files* or *Library files*:

- Require Files — These are source (text) files which are invoked via the REQUIRE declaration in the BLISS-32 source program.
- Library Files — These are special files created by the compiler in a previous library compilation and are invoked by the LIBRARY declaration in the BLISS-32 source program.

Since Library files are "pre-compiled," lexical processing and declaration parsing and checking need not be repeated each time these files are included in a compilation; hence their usage implies a considerable reduction in total compilation time.

The contents of Require files must be fully processed each time the file is used in a compilation. Hence, using Require files will, in general, be less efficient than using Library files. However, since these files operate under a somewhat less stringent set of syntactical rules, their usage may be warranted in situations where a higher level of flexibility is desired.

Macros

BLISS-32 provides an extensive macro building facility allowing frequently used groups of declaratives or expressions to be expressed in an abbreviated way. Macros are defined via MACRO and KEYWORDMACRO declaratives and are accessed by simple call statements. They are fully expanded at compile time. BLISS-32 allows parameters to be specified in the macro definition, thus allowing each block of text to be specialized by the actual parameters passed to it. Macros may be simple, iterative, or conditional expressions.

Debugging

The BLISS-32 compiler produces a list of error messages showing the source program line on which the error occurred followed by a description of the error. If the error is recoverable, then the compiler will generate a "warning" diagnostic and continue with the compilation process. If the error is serious enough to invalidate the compiler's internal representation of the module, then an "error" diagnostic is generated, and processing ceases following the syntax checking — no object module is produced.

If an error occurs at execution time, then BLISS-32 can access the VAX-11 Symbolic Debugger program. This program may be accessed when the object module is linked with the Debug option. The Debug program allows the programmer to examine and deposit values in storage, set breakpoints, call routines, trace through a program as it executes, and perform other operations useful in checking out a program. (See section on Symbolic Debugger for a further description of VAX-11 debugging facilities.)

Transportability Features

The BLISS-32 language is designed to facilitate transportability; that is, the writing of programs that can be executed on architecturally different machines with little or no modification. Several language features enhance transportability:

- Higher-level language elements such as IF-THEN-ELSE, DO, SELECT, etc., may be transferred from one machine to another with little alteration.
- Machine-specific functions can be separated from the mainline code via modularization, macros, and Library and Require files.
- Parameterization allows machine-specific information to be passed to BLISS-32 data structures.

BLISS's transportability makes it an ideal language for system programming applications — and a viable alternative to assembly language coding in those applications in which extreme machine dependence is not involved.

VAX-11 SORT

VAX-11 SORT is a utility and subroutine package designed to take advantage of the large addressing of the VAX/VMS operating system. It allows the user to reorder data from an input file into a new file in a sequence based upon control or key fields within the input records. SORT can be used as a separate utility invoked from a terminal, a batch stream, or as a set of subroutines callable from the VAX-11 languages.

SORT provides four different sorting techniques:

- RECORD — Record sort produces a reordered file sorted by specified key fields (that is, entire records are reordered). A record sort uses any VAX/VMS input device and can process any valid VAX-11 RMS format.
- TAG — Tag sort produces a reordered file by sorting only the record keys. SORT then randomly reaccesses the input file to create a resequenced output file according to those record keys. The tag sort method conserves temporary storage, but can accept only input files residing on disk.
- ADDRESS — Address sort produces an address file without reordering the input file. The address file, sorted by record keys, can later be used as an index to read the data base in the desired sequence. Any number of address files may be created for the same data base. A customer master file, for instance, may be referenced by customer-number index or sales territory index for different reports. This is the fastest of the four sorting methods.
- INDEX — Index sort produces an address file containing the key field of each data record and a pointer to its location in the input file. The address file can be used by programs to randomly access data from the original file in the desired sequence. Like address sort, this is a high-speed process.

VAX-11 SORT Performance Features

VAX-11 SORT can:

- produce reordered data files in ascending or descending order using up to 10 key fields
- sort sequential, relative, and indexed files
- sort fixed, variable, and VFC (variable with fixed control) records

- sort Character (in ASCII collating sequence), Decimal, and Binary data types
- sort input files from any VAX/VMS input device
- write sorted data files on any VAX/VMS output device
- determine its own work file requirements based on input file characteristics
- be called from COBOL, FORTRAN, and MACRO

Command String and Specification File

VAX-11 SORT is invoked via the VAX/VMS Digital Command Language (DCL) command SORT. The command statement contains a string of qualifiers which control the sort operation. These qualifiers indicate the type of sort to be performed (RECORD is the default), define sort keys, specify the number of work files to be used, and describe format, organization, and other characteristics of input and output files.

Use of the /SPECIFICATION: qualifier allows the the SORT utility to be controlled by sort specification statements contained in a separately defined specification file. This option allows dynamic program control of specification file statements, and therefore dynamic control of subsequent sorts using the same specification file modified. Also, specification file libraries can be maintained for often used sorts.

Sort As a Callable Routine

VAX-11 SORT can be called by COBOL, FORTRAN, and MACRO. Two interfaces may be used, File I/O Interface and Record I/O Interface. The File I/O Interface allows the user to specify an input file and an output file to SORT. SORT then reads the data from the input file and sorts it into the output file. The Record I/O Interface allows the user to pass each individual record to SORT. SORT then orders them and passes each record back in the correct order, individually, to the calling program. Both interfaces share a common set of routines, and the same calls are used for all languages. These calls and their functions are shown in Figure 7-10.

Routine Name	Function
1. SOR$INIT_SORT	Initialize scratch files, work area, sorting parameters.
2. SOR$PASS_FILES	Pass a file name to SORT.
3. SOR$RELEASE_REC	Pass a record to SORT.
4. SOR$SORT_MERGE	Initiate sorting and inter mediate merging of records.
5. SOR$RETURN_REC	Initiate final merge pass and receive output record from SORT.
6. SOR$END_SORT	Allow clean-up of files and work area to complete the sort operation.

Figure 7-10
Sort Call Routines

Sort Statistics

A set of statistics is printed automatically at the completion of each sort operation. These consist of elapsed execution time, the number of records read, sorted and output, the longest record length, the multiblock count used and the multibuffer count used for input and output, the merge order, the number of merge passes, the working set size used, the number of initial runs and the virtual memory used for the sort tree.

In addition, SORT statistics include statistics kept by VAX/VMS for the number of buffered and direct I/O operations, CPU time, and the number of page faults. Figure 7-11 illustrates a typical sort statistics printout.

These statistics can be used to tune the parameters used in a particular sort, such as working set size, number of work files, and other variables.

```
SORT STATISTICS*
            RECORDS READ: 10000
            RECORDS SORTED: 10000
            RECORDS OUTPUT: 10000
            LONGEST RECORD LENGTH: 80
            INPUT MULTI BLOCK COUNT: 11
            INPUT MULTI BUFFER COUNT: 2
            OUTPUT MULTI BLOCK COUNT: 20
            OUTPUT MULTI BUFFER COUNT: 2
            ORDER OF MERGE: 7
            NUMBER OF MERGE PASSES: 2
            NUMBER OF INITIAL RUNS: 38
            MAXIMUM WORKING SET USED: 128
            DYNAMIC VIRTUAL MEMORY ADDED: 236032
            ELAPSED TIME: 00:01:26:40

            BUFFERED IO COUNT: 23
            DIRECT IO COUNT: 227
            CPU TIME: 5055
            PAGE FAULTS: 15596
```

Figure 7-11
Sort Statistics Listing

PDP-11 DATATRIEVE/VAX

PDP-11 DATATRIEVE/VAX is a software product that provides users with the ability rapidly to extract and display data from RMS-11 files; it operates in PDP-11 compatibility mode under VAX/VMS. DATATRIEVE's facilities include the following:

- Inquiry and Update capability — allowing interactive data retrieval, sort, and file modification
- Report generation — producing summary and detailed reports, headings, footnotes, group totals, and report totals
- Data Definition — allowing creation, maintenance, and access of a data dictionary that defines RMS records and DATATRIEVE command procedures

DATATRIEVE operates on sequential, indexed, or relative files, and will perform selective retrieval on keyed or non-keyed fields.

Data Structures

PDP-11 DATATRIEVE/VAX uses a set of data structures called "record structures," "domains," and "collections."

Record structures describe the formats of records in particular files and are similar to data definitions in COBOL. (A typical record structure is shown in Figure 7-12.) Domains are named groups of data containing records of a single type. When using DATATRIEVE with RMS-11, domains and files may be regarded as synonymous; DATATRIEVE will always refer to the domain name. Collections are subsets of domains consisting of records which share common characteristics (e.g., all employees with incomes between $15,000 and $20,000); collections may consist of no records, a single record, or many records, up to and including the entire domain. DATATRIEVE operates on collections of records from RMS-11 files rather than on the files themselves.

```
QL>SHOW YACHT
RECORD YACHT
USING
01 BOAT
03 TYPE.
   06 MANUFACTURER PIC X(10)
   QUERY-NAME IS BUILDER.
   06 MODEL PIC X(10)
03 SPECIFICATIONS
   QUERY NAME SPECS.
   06 RIG PIC X(6).
      QUERY-NAME IS LOA.
```

Figure 7-12
Typical DATATRIEVE Record Structure

Data definitions are maintained by DATATRIEVE in the Data Dictionary. PDP-11 DATATRIEVE/VAX also provides commands to list the contents of the Data Dictionary, to delete entries, and to control access to individual entries in the Data Dictionary.

Command Language

In order to perform its query, file maintenance, report writing and other activities, PDP-11 DATATRIEVE/VAX uses a special command language. The commands are simple, follow an English-type syntax, and are designed to be learned easily by individuals with little or no programming experience. Typical command statements might be

FIND SCREWS WITH PART-NO = 223,

utilizing the FIND command, or

SELECT 7,

which utilizes the SELECT command to select the seventh record in the collection.

PDP-11 DATATRIEVE/VAX also provides more sophisticated commands such as the IF THEN ELSE sequence, and the BEGIN...END block, which allows groups of commands to be combined into a single compound command statement.

DATATRIEVE Inquiry and Update Facilities

PDP-11 DATATRIEVE/VAX's Inquiry and Update facilities include the following features:

- Establishing of collections via the FIND command. Operators provided include equal, not-equal, greater-than, less-than, greater-equal, less-equal, between, and, or, and not.
- Full file maintenance capabilities, allowing the user to create new records, modify existing records, or delete old records.
- Ability to reorder a collection via the SORT command.
- Ability to select a single record in a collection via the SELECT command.
- Ability to print the records of a collection via the PRINT command.

DATATRIEVE Report Writer Facilities

PDP-11 DATATRIEVE/VAX allows the user great discretion in specifying formats and contents of printed reports. In particular, the user can specify:

- report/page headers and trailers
- multiple control breaks
- automatic statistical functions (MAX, MIN, AVERAGE, TOTAL, COUNT)
- page and line limitations
- control headings and footings
- page totals, control totals, or report totals
- detail or summary reports

DATATRIEVE inquiry and report writer statements may be intermixed during a given session, the only restriction being that an inquiry statement must not occur within a report writer sequence.

Data Protection

The PDP-11 DATATRIEVE/VAX user can regulate access to domains, records, and procedures through access requirements recorded with the definitions in the Data Dictionary. There are five types of access, called the user's "privileges." These privileges are defined as follows:

- READ: User can only retrieve this resource (i.e., "read" the domain, record, or procedure).
- EXTEND: User can only add records to this resource.
- MODIFY: User can retrieve or change records in this resource.
- WRITE: User can do all of the above and can also erase records.
- CONTROL: User can issue data protection commands.

DATATRIEVE controls access based upon the user's password and UIC (user identification code). Hence one user might have the ability to modify a particular domain, while another might be permitted only to read the records in that domain.

VAX-11 PROGRAM DEVELOPMENT TOOLS

The VAX-11/780 program development tools include text editors, a linker, a librarian, a common run-time procedure library, and a debugger. These tools are available to the programmer through the VAX/VMS command language, as are the language compilers themselves.

The text editors can be used to create memos, documentation, and data files, as well as source program modules for any language processor. The linker, librarian, debugger, and run-time procedure library described below are used only in conjunction with the language processors that produce native code. The language processors that produce compatibility mode code offer their own task building, library, and debugging facilities, and they each include their own object time system libraries.

Editors

The programmer can use either or both of two text editors: SOS and SLP. SOS is an interactive text editor that enables the programmer to create and modify text files using commands entered from either a hard-copy or video terminal. The user can insert, delete, and replace lines, find and substitute strings, or modify the text a character at a time. Lines can be identified by line number, relative position, or by contents. An adjacent group of lines can bɛ copied or transferred from one place to another. Editing can be done in any order in the file. Editor parameters can be set to user-specified values and the current values can be shown. User-specific parameters can be set automatically at editor startup.

SLP is a programmed text editor that enables a user to modify an existing file by supplying a command file containing a list of the modifications to be made. The command file provides a reliable way to duplicate the changes made to a file at a later time or on another system. SLP provides a formal record of changes made to files, both in the source file and in an audit trail listing — a feature useful in tracking the stages of large programming projects.

Linker

The VAX/VMS linker accepts one or more native object modules produced by an assembler or compiler, resolves the symbols and procedure references between them, and produces an executable program image. The VAX/VMS linker also enables a programmer to create sharable images that can be linked subsequently with other modules to produce an executable image. Furthermore, the linker not only accepts object modules to produce executable or sharable images, it can also accept object module libraries, sharable images, and sharable image libraries.

The linker's ability to produce and use sharable images reduces program development time in much the same way as its ability to produce and use previously compiled object modules and object module libraries: it saves link time itself, since internal references in sharable images have been previously resolved, and it enables a programmer to take advantage of sharable code at both link time and run time. (For more information on sharable images, refer to the Operating System section.)

The linker has several other distinguishing features:

- It has no knowledge of how physical memory is allocated. Its concern is allocation of virtual addresses to an image.
- It keeps both an image file and the amount of virtual address space an image requires as small as possible.
- It allows the programmer to separate private code and data from potentially sharable code and data.
- It allows the programmer to separate code from read-only data and read/write data so that memory protection can be used to prevent and isolate programming errors.
- It allows both weak definitions and weak references for global symbols. Weak definitions and weak references are especially useful in creating libraries and language processors that use libraries.

Librarian

The librarian enables a programmer to create, update, modify, list and maintain library files. A library file can be a collection of object modules or sharable images. A programmer can request the linker to use one or more library files from which the linker can obtain modules to resolve references during linking.

Common Run-time Procedure Library

The run-time procedure library is a collection of general purpose and language-specific libraries available to any native program, regardless of the source language in which the program was written. The run-time library is a sharable image that allows:

- the choice of incorporating procedures from the library into an executable image, or mapping the global sections into a process virtual address space at run time
- a single copy of the library to be shared by all processes
- installation of a new library without the need to relink existing programs

The run-time library includes the following:

- a mathematical library
- a general utility library
- a condition handling facilities library
- a language-independent support library
- a FORTRAN IV-PLUS language specific support library

Symbolic Debugger

The debugger can be linked with a native program image to control program execution during development. The debugger can be used interactively or it can be controlled from a command procedure file. The debugging language is similar to the VAX/VMS command language. Expressions and data references are similar to those of the source language used to create the image being debugged. Debugging commands include the ability to start and interrupt program execution, to step through instruction sequences, to call routines, to set break or trace points, to set default modes, to define symbols, and to deposit, examine, or evaluate virtual memory locations.

RSX-11M PROGRAM DEVELOPMENT

Software bundled within the operating system allows VAX/VMS users to write, assemble or compile, and link RSX-11M task images. The task images can be written to execute on an RSX-11M or RSX-11S system or, if properly coded, they can be written to execute in the VAX-11/780 compatibility mode environment. (Refer to the Processor and Operating System sections for descriptions of compatibility mode.)

Programming Languages

The user can develop RSX-11M programs in several languages. The PDP-11 MACRO assembler, included in the operating system, makes RSX-11M executive directives and the PDP-11 instruction set directly available. In addition, the the PDP-11 BASIC-PLUS-2/VAX and PDP-11 COBOL/VAX compilers can be used to create task images for execution on RSX-11M or RSX-11S systems.

The FORTRAN-IV/VAX Cross-Development Compiler is an optional product which allows users to develop

FORTRAN IV programs for the RSX-11M and RSX-11S systems. The FORTRAN IV language allows the programmer to use RSX-11M directives. FORTRAN IV object modules can be linked with other FORTRAN IV or PDP-11 MACRO object modules into executable task images. FORTRAN IV includes an object time system (a library of commonly used FORTRAN routines such as math, error handling, and process I/O routines) as well as the RSX-11M executive directives.

RSX-11M Program Development Tools

VAX programmers can use the RSX-11M MCR command interface on the VAX-11/780 system. MCR enables the programmer to run the standard RSX-11M utilities such as the EDI and SLP editors, the PIP and FLX file transfer utilities, as well as the task builder, librarian, and patch utilities.

The RSX-11M task builder creates loadable task images from object modules created by the PDP-11 MACRO assembler and/or PDP-11 FORTRAN IV compiler, the PDP-11 COBOL compiler, or the PDP-11 BASIC-PLUS-2 compiler. It links relocatable object modules and resolves any references to global symbols, common areas, and shared libraries. The RSX-11M librarian provides the user with the ability to create and maintain disk-resident libraries of object modules and user-defined macros.

The RSX-11M on-line debugger (ODT) can be linked with PDP-11 MACRO or FORTRAN IV task images to aid in debugging programs. The RSX-11M ZAP task patch utility is used to examine and modify task image files and data files. With ZAP, permanent patches can be made to task image or data files without having to re-create the file. The RSX-11M PAT object module patch utility allows the patching or updating of code in a relocatable binary object module.

BLANK

8 The Data Management Services

```
ENVIRONMENT DIVISION.
[INPUT-OUTPUT SECTION.]
FILE-CONTROL.
SELECT file-name
   ASSIGN TO device-name-1   [, device-name-2]  . . .
   ; ORGANIZATION IS INDEXED
 [                  ( SEQUENTIAL ) ]
 [ ; ACCESS MODE IS { RANDOM     } ]
 [                  ( DYNAMIC    ) ]
   ; RECORD KEY IS data-name-1
 [ ;ALTERNATE RECORD KEY IS data-name-2  [WITH DUPLICATES] ] . . .

DATA DIVISION.
[FILE SECTION.
[FD file-name
 [ ; BLOCK CONTAINS  [integer-1 TO]  integer-2  { RECORDS    } ]
 [                                              { CHARACTERS } ]
 [ ; RECORD CONTAINS  [integer-3 TO]  integer-4 CHARACTERS ]
   ; LABEL { RECORD IS   } { STANDARD }
           { RECORDS ARE } { OMITTED  }
 [ ; DATA   { RECORD IS   }  data-name-3 [, data-name-4] . . . ]
 [          { RECORDS ARE }                                    ]

PROCEDURE DIVISION.
       ( INPUT file-name-1  [, file-name-2]  . . . )
OPEN   { OUTPUT file-name-3  [, file-name-4]  . . . } . . .
       ( I-O file-name-5   [, file-name-6]  . . . )
```

VAX/VMS data management includes a file system that provides volume structuring and protection, and record management services that provide device-independent access to the VAX-11/780 peripherals.

The VAX/VMS on-disk structure provides a multiple-level hierarchy of named directories and subdirectories. Files can extend across multiple volumes and be as large as the volume set on which they reside. Volumes are mounted to identify them to the system. VAX/VMS also supports multi-volume ANS format magnetic tape files with transparent volume switching.

The VAX/VMS record management input/output system (RMS) provides device independent access to disks, tapes, unit record equipment, terminals, and mailboxes. RMS allows users and application programs to create, access, and maintain data files with efficiency and economy. Under RMS, records are regarded by the user program as logical data units that are structured and accessed in accordance with application requirements.

RMS provides sequential record access to sequential file organizations, and sequential, random, or combined record access to relative file organizations. Compatibility mode application programs can also process multi-key indexed files sequentially, randomly, or in combination using index keys. Multi-key indexed file processing includes incremental reorganization.

COMPONENTS

The operating system's data management services are provided by the:

- device drivers
- file system
- record management services
- command interpreter and utility programs

The device drivers provide the basic I/O device handling for all of the other data management services. Device drivers and their features are described in the Peripherals and Operating System sections.

The file system provides volume structuring and directory access to disk and magnetic tape files. Programmers can use the file system as a base to build their own record processing system, or they can use the VAX/VMS record management services.

The record management services (RMS) provide device-independent access to all types of I/O peripherals. The RMS procedures enable a program to access records within files, and provide the same programming interface regardless of device characteristics. The system includes utilities for RMS file creation and maintenance.

As described in the Users section, the command interpreter enables a user to reserve devices for exclusive use, set device and directory name defaults, and assign logical names to file specifications. The command interpreter also enables the user to execute file management utilities that provide file copy, transfer, and conversion operations.

The following paragraphs discuss some of the features and functions of the file system, including the file structures, file naming facilities, and the file management utility programs. The remainder of this section describes the record management services programming environment.

FILE MANAGEMENT

VAX/VMS provides two file structures: one for disk volumes and one for magnetic tape volumes. From the user's point of view, the only differences between the two file structures are those imposed by the capabilities of the media. Volumes are mounted for identification, and files can extend across multiple volumes. The practical limit to file size is that they can be only as large as the volume set on which they reside.

Volume and file protection is based on User Identification Codes (UICs) assigned to accessors and the file or volume. The UICs establish the accessor's relationship to the data structure as Owner, the owner's Group, the System, or the World (all others). Depending on the relationship, the accessor may or may not have read, write, execute, or delete access to any given file.

Disk volumes are multi-user volumes. They can contain a multi-level directory hierarchy that is defined dynamically by the users of the volume. The on-disk file structure appears to a program to be a virtually contiguous set of blocks. The blocks of the file, however, may be scattered anywhere on a volume. Mapping information is maintained to identify all the blocks constituting a file. Figure 8-1 illustrates the file structure.

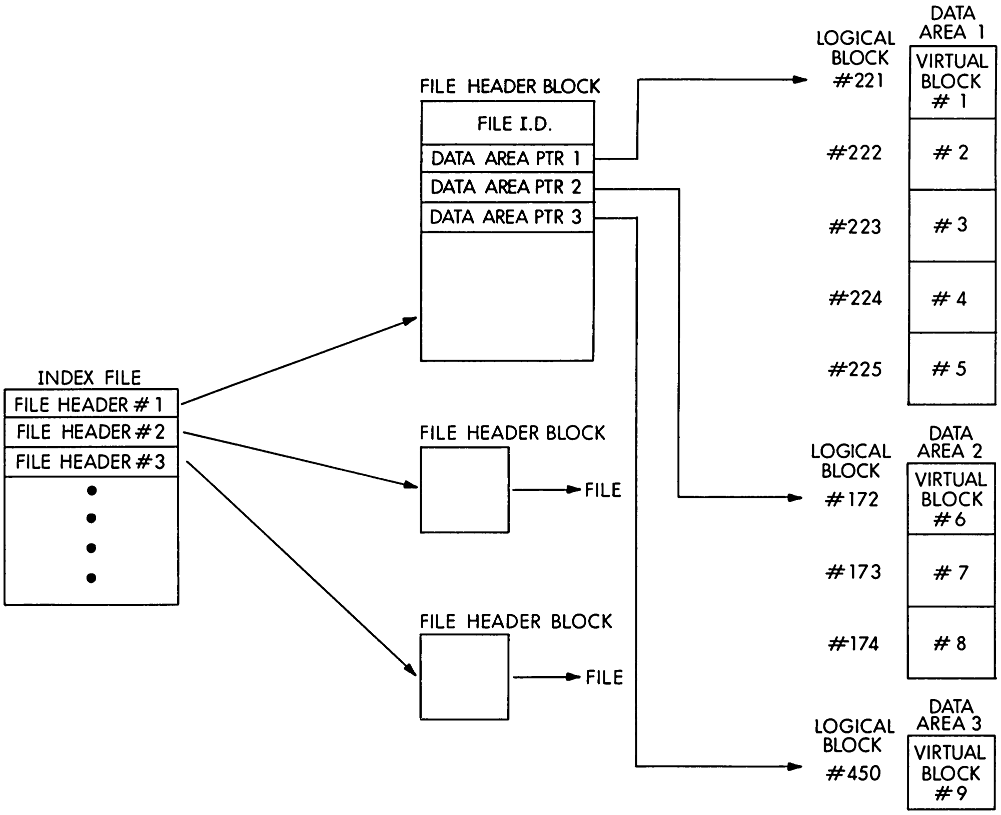


Figure 8-1
Disk File Structure

Disk files can be extended easily. The blocks of the file are allocated in physically contiguous sets, called **extents**. Users are not required to preallocate space, although they can do so. Users can specify placement on an allocation request, and they can control automatic allocation. For example, when a file is automatically extended, it can be extended by any given number of contiguous blocks. If desired, a file can be created as a contiguous file, in which case it is both virtually and physically contiguous.

The disk structure includes duplicates of its critical volume information. The system detects bad disk blocks dynamically and prevents re-use once the files to which they are allocated are deleted.

Magnetic tape volumes are single-user volumes. Magnetic tape files consist of physically contiguous blocks. Record blocking is under program control. Files have ANS format labels. VAX/VMS also supports unlabeled (non-file structured) magnetic tapes.

File Directories and Directory Structures

A directory is a file containing a list of files on a given volume. A directory entry contains the name, type, version, and unique file ID for a particular file. A directory can list files having the same owner UIC or files having different owner UICs. The entries are listed alphabetically.

A disk volume contains at least one directory, called the master file directory. The system manager is responsible for creating a volume's master file directory. The master file directory can (and normally does) contain a list of directory files which form a second level of directories. The second level of directory files can list data files and/or other directory files, called **subdirectories**. Users can create subdirectories within the directories they own. The subdirectories can also list other directory files and/or data files. Figure 8-2 illustrates a multi-level directory structure.

Since directories of files on volumes are files themselves, they are assigned owner UICs and can be protected from certain kinds of access depending on the relationship established by an accessor's UIC. In the special case of directory files, the file protection fields control an accessor's ability to:

- look up files
- enter new files in the directory, including new versions of existing files
- remove files from the directory

File Specifications

A **file specification** identifies which file is to be used in a file processing operation. Programs use file specifications to identify the file they want to create, access, delete, or extend, and users supply the command interpreter with a file specification to identify the file they want to edit, compile, copy, or delete, etc. A complete file specification is a well-defined character string composed of the following fields:

- *Node Name* — The node of the network in which the volume containing the file is stored. The node name is followed by two colons (::) to delimit it from the remainder of the file specification.
- *Device Name* — The device on which the volume containing the file is mounted. The device name is followed by a single colon (:) to delimit it from the remainder of the file specification.
- *Directory Name* — The directory in which the file is listed. A directory name begins with an opening bracket (< or [) and ends with a closing bracket (> or]). If the file is listed in a subdirectory, the directories to be searched are listed in the desired search order, with the names separated by periods, e.g.: [name1.name2.name3]

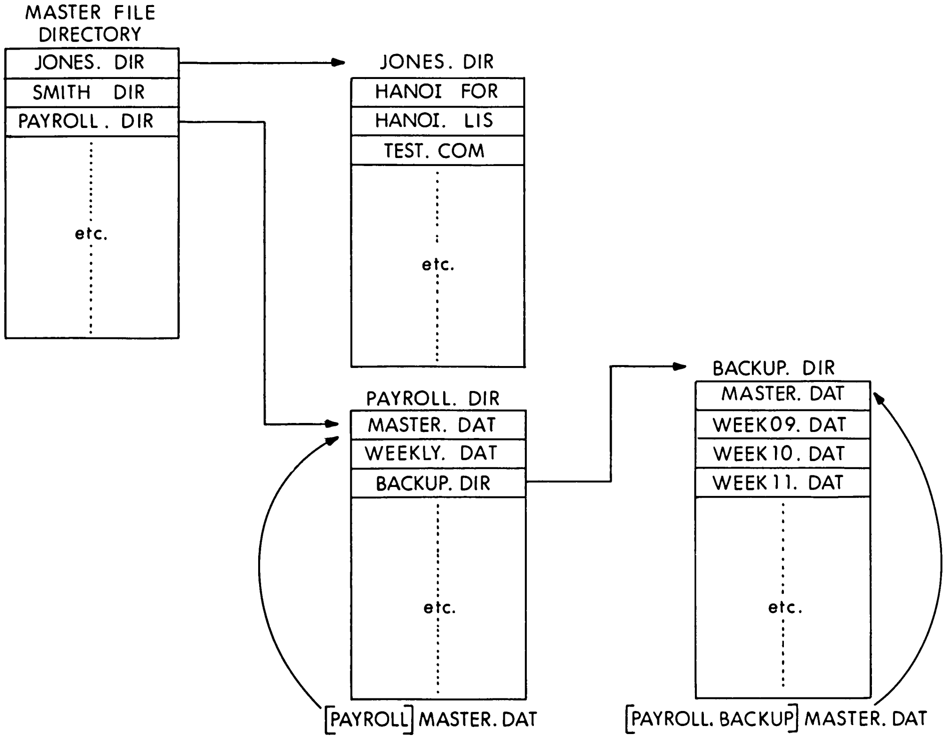


Figure 8-2
Multi-Level Directory Structure

- *File Name* — The user-assigned name of the file.
- *File Type* — The type identification for the file. The type is preceded by a period (.) to delimit it from the remainder of the file specification.
- *File Version* — the generation number of the file. The file version is preceded by a semicolon (;) or period (.) to delimit it from the remainder of the file specification.

For example, a complete file specification might be:

NODE47::DBA1:[JONES]HANOI.FOR;2

In this case, NODE47 is the name of the network node, DBA1 is the name of the device (DB for disk pack device, A for disk controller, 1 for drive unit number), [JONES] is the directory name, HANOI is the file name, FOR is the file type (meaning that the file is a FORTRAN source file), and 2 is the version number.

Neither programs nor command language users need to provide a complete file specification to identify files. The system applies defaults to most fields of a file specification when they are not present. For example, if the node name is not present, the node is assumed to be the node on which the program is executing. If the version number is not present, the version is always assumed to be the latest version. Device name and directory name defaults for users and the programs they execute are supplied by the system manager in the user authorization file, and users can change the standard defaults at any time during their session on the system.

Some commands (such as COPY, PRINT, and DELETE) accept a wild card in one or more fields of a file specification. A wild card is an asterisk appearing in a file specification field that means "all."

File specifications also apply to non-file structured devices such as line printers, card readers, and terminals. In these cases, however, the user or program needs to supply only the node name and device name, as appropriate.

Logical File Naming

To provide both system and device independence, users and programs are not limited to identifying files by their file specifications. They can use **logical names** in place of a complete file specification, or in place of a portion of a file specification. For example, a user can assign a logical name to the left-most three fields of a file specification:

$ ASSIGN NODE47::DBA4:[JONES] *to* VOL

And then use the logical name VOL in a subsequent command:

$ TYPE VOL:HANOI.FOR

Defaults also apply when translating logical names, so that the user could have made the assignment:

$ ASSIGN NODE47::[JONES] *to* VOL

In this case, the user's default device name would be used to derive the complete file specification.

Logical name assignments can be made on a process, group, or system wide basis. Logical names can also be recursive, that is, a logical name can be assigned to another logical name, or to a logical name and a portion of a file specification.

For example, suppose a company's weekly payroll production run includes an application program that uses the current week's payroll changes data file. That data file may be located in the directory named [PAYROLL] one week, or in the payroll backup subdirectory, [PAYROLL.BACKUP], another week. The volume on which the file is stored may be mounted on disk pack drive unit number 1 one week, or on unit 2 another week.

The application programmer can write the program without knowing which directory the data file is listed in, or which device the volume is mounted on. A series of logical name assignments provides the complete file specification. The assignments are the responsibility of the people who know what directory the file is listed in, and what drive the volume is mounted on.

In the example shown in Figure 8-3, the application program contains an OPEN statement for the payroll data file using the logical name WEEKLY_PAYROLL_CHANGES (note that underscore is a legal character). The application systems designer has created a command procedure file

Application Programmmer:

```
OPEN ("WEEKLY_PAYROLL_CHANGES")
```

Application System Programmer:

```
Command Procedure: PAY_RUN.COM
accepts one parameter (P1): Week Number
$ ASSIGN WEEKLY_PAYROLL:'P1'.WPY     WEEKLY_PAYROLL_CHANGES
$ RUN APPLICATION
```

Production Clerk:

```
$ @PAY_RUN WEEK09
```

Payroll Group Operations Manager:

```
$ ASSIGN/GROUP PAY_PACK:[PAYROLL.BACKUP]     WEEKLY_PAYROLL
```

Local Operator:

```
$ ASSIGN/SYSTEM DBA2:     PAY_PACK
```

Figure 8-3
Logical Name Assignment

called PAY_RUN that controls the production run. The command procedure file includes a logical name assignment that obtains the file name as a parameter supplied by the operator or production clerk who starts the production run. The logical name used by the application program is given a value that consists of another logical name (WEEKLY_PAYROLL) and the file name and type specifications.

To complete the series of logical name assignments, the payroll group operations manager makes a group-wide logical name assignment: the payroll data files this week are stored in the PAYROLL.BACKUP subdirectory. The logical name assignment provides the directory name, using another logical name (PAY_PACK) known to the operator who mounts the payroll data files volume. The operator makes the system-wide logical name assignment when mounting the pack before the production run. Given the assignments shown in the example, the logical name used to open the file is translated to:

DBA2:[PAYROLL.BACKUP]WEEK09.WPY

(The local system node name and the latest version number are used as defaults to complete the file specification.) Should the directory name change, or the pack be mounted on another device that day, the only changes made are the logical name assignments. There is no need to modify either the application program or the command procedure controlling the production run.

File Management

The VAX/VMS system includes many services that aid in data management and maintenance. Some of these are described in the following paragraphs.

Sorting Files

The SORT program allows the user to rearrange, delete, and reformat records in a file. The user can arrange the records in the ascending or descending sequence of one or more fields within the records for subsequent sequential processing. SORT can also create several different index files for accessing a file according to these indices without reordering the data itself. SORT provides four sorting techniques:

Record Sort produces a reordered data file by manipulating all records in their entirety. The data can be available on any acceptable input device: cards, magnetic tape, or disk. The records can be variable or fixed length.

Tag Sort produces a reordered data file by manipulating only the key fields used to order the records.

ADDROUT Sort produces an ordered address file that enables the user to access the records in that data file in the order of the address file.

Index Sort produces an ordered key file that can be used to sequentially or randomly access records in the data file.

SORT accepts two kinds of command formats: a simple keyboard-oriented command string, and a conventional specification file format. The latter format allows input record selection techniques based on user-defined record characteristics and field specifications. In addition, the input records can vary in format and the output records can be restructured.

Comparing Files

A file differences command contrasts two files by automatically aligning matching text, and optionally ignoring comments, empty records, trailing blanks, or multiple blanks. The output can be a file-by-file list of differences, an interleaved list of differences, a list with change bars, or a batch editor command input file.

Backing Up Files and Volumes

The Disk Save and Compress (DSC) utility enables a user to backup entire disk volumes to magnetic tape or to other disks. When backing up disk volumes to other disk volumes, or restoring disk volumes from magnetic tape, DSC combines unused blocks on disks into contiguous areas.

Verifying File Structures

The file verification utility checks the consistency and accuracy of the file structure on a Files-11 disk volume. It can also display the number of available blocks in a volume, locate files that could not otherwise be accessed, and list the names of files on the volume.

Bad Block Locator

The bad block locator utility determines the number and location of bad blocks on Files-11 disk volumes and stores this information in the bad block file on the volume so that the blocks can not be allocated. Running this utility before initializing a Files-11 volume is useful in ensuring a disk's integrity.

RMS Utilities

The record management services procedures are complemented by a number of utilities designed especially for RMS file creation and maintenance. They allow the user to:

- Create an RMS file and define the attributes of the file.
- List the attributes of a single file or a group of files, or list the contents of a backup magnetic tape.
- Convert a file with any file organization or record format to a file with any other file organization or record format.
- Backup a single file or group of files in a compact format (optionally by creation or revision date).
- Restore files previously backed up (optionally by creation or revision date).

RECORD MANAGEMENT SERVICES

The record management services (RMS) are a set of system procedures that provide efficient and flexible facilities for data storage, retrieval, and modification. When writing programs, the user can select processing methods from among the RMS file organizations and accessing techniques. The following sections discuss the RMS:

- file organizations
- file attributes
- program operations
- run-time environment

The manner in which RMS builds a file is called its organization. RMS provides three file organizations:

- sequential
- relative
- indexed

The sequential and relative file organizations can be processed using native and compatibility mode programming languages. The indexed file organization can be processed using compatibility mode programming languages only.

The sequential and relative file organizations can be processed using native and compatibility mode programming languages. The indexed file organization can be processed using compatibility mode programming languages only.

The organization of a file establishes the techniques one can use to retrieve and store data in the file. These techniques are known as record access modes. The record access modes that RMS supports are:

- sequential
- random
- Record's File Address (RFA)

An application program or a RMS utility can be used when creating a RMS file to specify the organization and characteristics of the file. Among the attributes specified are:

- storage medium
- file name and protection specifications
- record format and size
- file allocation information

After RMS creates a file according to the specified attributes, application programs can store, retrieve and modify data. These program operations take place on the logical records in a file or the blocks comprising the file.

RMS FILE ORGANIZATIONS

A file is a collection of related information. For example, a file might contain a company's personnel information (employee names, addresses, job titles). Within this file, the information is divided into records. All the information on a single employee might constitute a single record. Each record in the personnel file would be subdivided into discrete pieces of information known as fields. The user defines the number, locations within the record, and logical interpretations of these fields.

The user can completely control the grouping of fields into records and records into files. The relationship among fields and records is embedded in the logic of the programs. RMS does not know the logical relationships that exist within the information in the files.

RMS ensures that every record written into a file can subsequently be retrieved and passed to a requesting program as a single logical unit of data. The structure, or organization, of a file establishes the manner in which RMS stores and retrieves records. The way a program requests the storage or retrieval of records is known as the record access mode. The organization of a file determines which record access modes can be used.

Sequential File Organization

In sequential file organization, records appear in consecutive sequence. The order in which records appear is the order in which the records were originally written to the file by an application program or RMS utility. Sequential organization is the only file organization permitted for magnetic tape and unit record devices. Figure 8-4 illustrates sequential file organization.

Relative File Organization

When relative organization is selected, RMS structures a file as a series of fixed-size record cells. Cell size is based on the maximum length permitted for a record in the file. These cells are numbered from 1 (the first) to n (the last). A cell's number represents its location relative to the beginning of the file.

Each cell in a relative file can contain a single record. There is no requirement, however, that every cell contain a record. Empty cells can be interspersed among cells containing records. Figure 8-5 illustrates a relative file organization.

Since cell numbers in a relative file are unique, they can be used to identify both a cell and the record (if any) occupying that cell. Thus, record number 1 occupies the first cell in the file, record number 17 occupies the seventeenth cell, and so forth. When a cell number is used to identify a record, it is also known as a relative record number.

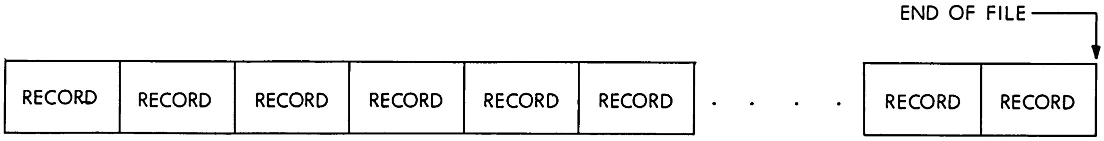


Figure 8-4
Sequential File Organization

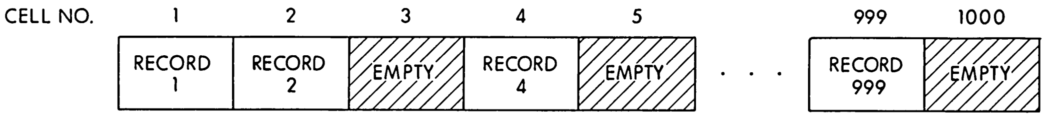


Figure 8-5
Relative File Organization

Indexed File Organization

The location of records in indexed file organization is transparent to the program. RMS completely controls the placement of records in an indexed file. The presence of keys in the records of the file governs this placement.

A key is a field present in every record of an indexed file. The location and length of this field are identical in all records. When creating an indexed file, the user decides which field or fields in the file's records are to be a key. Selecting such fields indicates to RMS that the contents (i.e., key value) of those fields in any particular record written to the file can be used by a program to identify that record for subsequent retrieval.

At least one key must be defined for an indexed file: the primary key. Optionally, additional keys or alternate keys can be defined. An alternate key value can also be used as a means of identifying a record for retrieval.

As programs write records into an indexed file, RMS builds a tree-structured table known as an index. An index consists of a series of entries containing a key value copied from a record that a program wrote into the file. Stored with each key value is a pointer to the location in the file of the record from which the value was copied. RMS builds and maintains a separate index for each key defined for the file. Each index is stored in the file. Thus, every indexed file contains at least one index, the primary key index. Figure 8-6 illustrates an indexed file organization with a primary key. When alternate keys are defined, RMS builds and stores an additional index for each alternate key.

RMS RECORD ACCESS MODES

The methods of retrieving and storing records in a file are called record access modes. A different record access mode can be used to process records within the file each time it is opened. A program can also change record access mode during the processing of a file. RMS permits only certain combinations of file organization and record access mode. Table 8-1 lists these combinations.

Table 8-1
Record Access Modes and File Organizations

File Organization	Record Access Mode			
	Sequential	Random		RFA
		Record #	Key Value	
Sequential	Yes	Yes[2]	No	Yes[1]
Relative[1]	Yes	Yes	No	Yes
Indexed[1]	Yes	No	Yes	Yes

[1] Disk files only.
[2] Fixed length record format disk files only.

Sequential Record Access Mode

Sequential record access mode can be used to access all RMS files and all record-oriented devices, including mailboxes. Sequential record access means that records are retrieved or written in the sequence established by the organization of the file.

Sequential Access to Sequential Files — When using sequential record access mode in a sequentially organized file, physical arrangement establishes the order in which records are retrieved. To read a particular record in a file, say the fifteenth record, a program must open the file and access the first fourteen records before accessing the desired record. Thus each record in a sequential file can be retrieved only by first accessing all records that physically precede it. Similarly, once a program has retrieved the fifteenth record, it can read all the remaining records (from the sixteenth on) in physical sequence. It cannot, however, read any preceding record without closing and reopening the file and beginning again with the first record.

Sequential Record Access to Relative Files — During the sequential access of records in the relative file organization, the contents of the record cells in the file establish the order in which a program processes records. RMS recognizes whether successively numbered record cells are empty or contain records.

When a program issues read requests in sequential record access mode for a relative file, RMS ignores empty record cells and searches successive cells for the first one con-

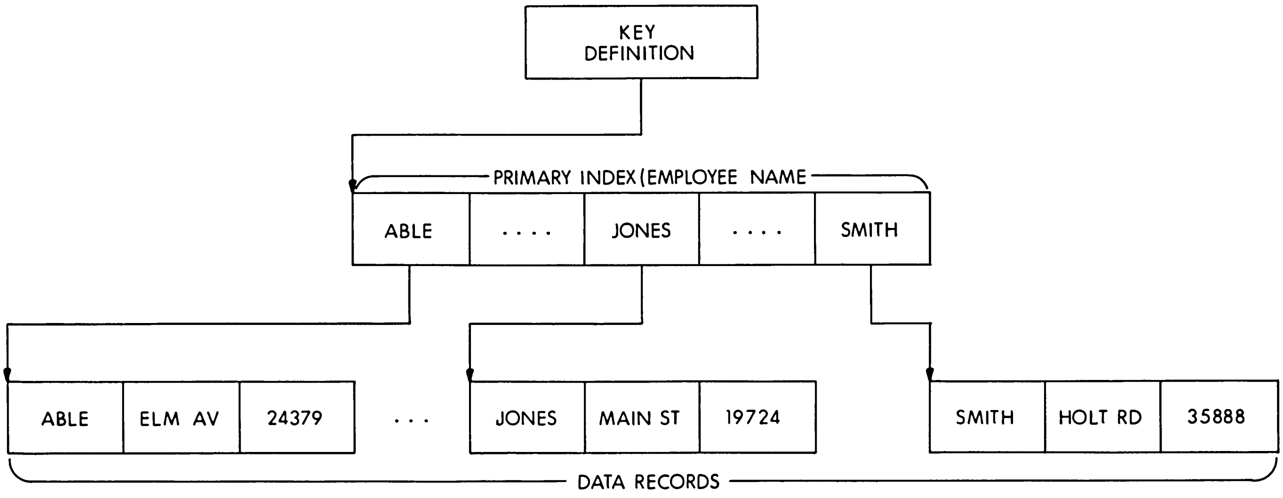


Figure 8-6
Indexed File Organization

taining a record. When a program adds new records in sequential record access mode to a relative file, RMS places a record in the cell whose relative number is one higher than the relative number of the previous request, as long as that cell does not already contain a record. RMS allows a program to write new records only into empty cells in the file.

Sequential Record Access to Indexed Files — A program can use the sequential record access mode to retrieve records from an indexed file in the order represented by any index. The entries in an index are arranged in ascending order by key values. If more than one key is defined for the file, each separate index associated with a key represents a different logical ordering of the records in the file.

When reading records in sequential record access mode from an indexed file, a program initially specifies a key (primary key, first alternate key, second alternate key, etc.) to RMS. Thereafter, RMS uses the index associated with that specified key to retrieve records in the sequence represented by the entries in the index. Each successive record RMS returns in response to a read request contains a value in the specified key field that is equal to or greater than that of the previous record returned.

When writing records to an indexed file, RMS uses the definition of the primary key field to place the record in the file.

Random Record Access Mode

In random record access mode, the program establishes the order in which records are processed. Each program request for access to a record operates independently of the previous record accessed. Each request in random record access mode identifies the particular record of interest. Successive requests in random mode can identify and access records anywhere in the file.

Random Record Access to Sequential Files — Native programs can access sequential files on disk using relative record number to randomly locate a record, provided that the records are in fixed-length record format.

Random Record Access to Relative Files — Programs can read or write records in a relative file by specifying relative record number. RMS interprets each number as the corresponding cell in the file. A program can read records at random by successively requesting, for example, record number 47, record number 11, record number 31, and so forth. If no record exists in a specified cell, RMS notifies the requesting program. Similarly, a program can store records in a relative file by identifying the cell in the file that a record is to occupy. If a program attempts to write a new record in a cell already containing a record, RMS notifies the program.

Random Record Access to Indexed Files — For indexed files, a key value rather than a relative record number identifies the record. Each program read request in random record access mode specifies a key value and the index (primary index, first alternate index, second alternate index, etc.) that RMS must search. When RMS finds the key value in the specified index, it reads the record that the index entry points to and passes the record to the user program.

Program requests to write records randomly in an indexed file do not require the separate specification of a key value. All key values (primary and, if any, alternate key values) are in the record itself. When an indexed file is opened, RMS retrieves all definitions stored in the file. RMS knows the location and length of each key field in a record. Before writing a record into the file, RMS examines the values contained in the key fields and creates new entries in the indices. In this way RMS ensures that the record can be retrieved by any of its key values.

Record's File Address (RFA) Record Access Mode

Record's File Address (RFA) record access mode can be used to retrieve records in any file organization as long as the file resides on a disk volume. Like random record access mode, RFA record access allows a specific record to be identified for retrieval, using the record's unique address. The actual format of this address depends on the organization of the file.

After every successful read or write operation, RMS returns the RFA of the subject record to the program. The program can then save this RFA to use again to retrieve the same record. It is not required that this RFA be used only during the current execution of the program. RFAs can be saved and used at any subsequent time.

Dynamic Access

Dynamic access is not strictly an access mode. It is the ability to switch from one record access mode to another while processing a file. For example, a program can access a record randomly, then switch to sequential record access mode for processing subsequent records. There is no limitation on the number of times such switching can occur. The only limitation is that the file organization must support the record access mode selected.

FILE AND RECORD ATTRIBUTES

When creating an RMS file, a program or user defines its logical and physical characteristics, or attributes. These characteristics are defined by source language statements in an application program or by an RMS utility. The program or user assigns the file a name, the owner's User Identification Code, and a protection code, and selects the file organization. The program or user also defines or selects other attributes, including:

- device
- file size
- file location
- record format and size
- keys (for indexed files only)

Selection of device is related to the organization of the file. Sequential files can be created on Files-11 disk volumes or ANS magnetic tape volumes. Sequential files can also be read from mailboxes, terminals, and card readers, and written to mailboxes, terminals, and line printers. Relative and indexed files can be created on Files-11 disk volumes.

The logical limit on file size is 2^{31-1} blocks, with a more realistic limit being the volume set on which a file can reside. When creating an RMS file on a disk volume, the user can specify an initial allocation size. If no file size is given, RMS allocates the minimum amount of storage needed to contain the defined attributes of the file. The initial size can be extended dynamically. The user can let RMS locate the file, or the user can allocate the file to specific locations on the disk to optimize disk access time. The file's starting location can be specified optionally using a volume-relative block number, or a physical cylinder address.

When creating a file on a magnetic tape volume, a user or program does not specify an initial allocation size. The blocks are simply written one after another down the tape, beginning after the last file, if any, written on the tape. Once a tape file has been created, another file can replace it or be appended to it, but all subsequent files on the tape, if any, are lost.

Record Formats

The user provides the format and maximum size specifications for the records the file will contain. The specified format establishes how each record appears in the file. The size specification allows RMS to verify that records written into the file do not exceed the length specified when the file was created.

Fixed length record format refers to records of a file that are all equal in size. Each record occupies an identical amount of space in the file. All file organizations support fixed length record format.

Variable-length record format records can be either equal or unequal in length. All file organizations support variable-length record format. RMS prefixes a count field to each variable-length record it writes. The count field describes the length (in bytes) of the record. RMS removes this count field before it passes a record to the program. RMS produces two types of count fields, depending on the storage medium on which the file resides:

- Variable-length records in files on Files-11 disk volumes have a 2-byte binary count field preceding the data field portion of each record. The specified size excludes the count field.
- Variable-length records on ANS magnetic tapes have 4-character decimal count fields preceding the data portion of each record. The specified size includes the count field. In the context of ANS tapes, this record format is known as D format.

Variable-with-fixed-control (VFC) records consist of two distinct parts, the fixed control area and a variable-length data record. Although stored together, the two parts are returned to the program separately when the record is read. The size of the fixed control area is identical for all records of the file. The contents of the fixed control area are completely under the control of the program and can be used for any purpose. For example, fixed control areas can be used to store the identifier (relative record number or RFA) of related records. Indexed file organizations do not support VFC record format.

Key Definitions For Indexed Files

To define a key for an indexed file, the user specifies the position and length of particular data fields within the records. At least one key, the primary key, must be defined for an indexed file. Additionally, up to 254 alternate keys can be defined. In general, most files have two or three keys. Because indices require storage space and RMS updates indices as records are added or modified, no more than 6 to 8 keys should be defined where storage space or access time is important.

Each primary and alternate key represents from 1 to 255 bytes in each record of the file. RMS permits 6 key field data types.

- string
- signed 15-bit integer
- unsigned 16-bit binary
- signed 31-bit integer
- unsigned 32-bit binary
- packed decimal

The string key field can be composed of simple or segmented keys. A simple key is a single, contiguous string of characters in the record; in other words, a single field. A segmented key, however, can consist of from 2 to 8 fields within records. These fields need not be contiguous. When processing records that contain segmented keys, RMS treats the separate fields (segments) as a logically contiguous character string. The integer, binary, and packed decimal data types can only be simple keys.

When defining keys at file creation time, two characteristics for each key can be specified:

- Duplicate key values are or are not allowed.
- Key value can or cannot change.

When duplicate key values are allowed, more than one record can have the same value in a given key. For example, the creator of a personnel file could define the department name field as an alternate key. As programs wrote records into the file, the alternate index for the department name key field would contain multiple entries for each key value (e.g., PAYROLL, SALES, ADMINISTRATION), since departments are composed of more than one employee.

When such duplication occurs, RMS stores the records so that they can be retrieved in first-in/first-out (FIFO) order.

If key values can change, records can be read and then written back into the file with a modified key value. For example, this specification would allow a program to access a record in the personnel file and change the contents of a department name field to reflect the transfer of an employee from one department to another. This characteristic can be specified only for alternate keys. If key values can change, the user must also specify that the duplicate key values are allowed. If the primary key value can change, the user may not change the record length.

Figures 8-7 and 8-8 show excerpts from a COBOL program which operates upon an indexed customer information file via the dynamic access method. The program searches through the file and generates various reports based upon the customer's financial status and additional input typed in by the user at the terminal. In Figure 8-7, the program describes the organization of the file and specifies the access method to be used. In Figure 8-8, the program searches for the first non-zero customer number. Using the "approximate key" match facility (greater than), the program searches for the first non-zero customer. When RMS has located the first non-zero customer number, the program changes access method and the file is read sequentially.

```
INPUT-OUTPUT SECTION.

FILE-CONTROL.

SELECT CUSTOMER-FILE
     ASSIGN TO "CUSTOM.DAT"
     ORGANIZATION IS INDEXED
     ACCESS MODE IS DYNAMIC
     RECORD KEY IS CUST-CUSTOMER-NUMBER
     ALTERNATE RECORD IS KEY IS CUST-CUSTOMER-NAME
     FILE STATUS IS CUSTOMER-FILE-STATUS.
```

Figure 8-7
ISAM File Description

```
      OPEN INPUT CUSTOMER-FILE.
      MOVE "000000" TO CUST-CUST-NUMBER.
      START CUSTOMER-FILE
            KEY IS > CUST-CUST-NUMBER.
      OPEN OUTPUT STATEMENT-REPORT.

************************************

MAINLINE SECTION.
SBEGIN.
      READ CUSTOMER-FILE NEXT
            AT END
               GO TO END-PROCESS.
      ADD 1 TO RECORD-COUNT.
*
```

Figure 8-8
Dynamic Access Processing

PROGRAM OPERATIONS ON RMS FILES

After RMS has created a file according to the user's description of file characteristics, a program can access the file and store and retrieve data.

When a program accesses the file as a logical structure (i.e., a sequential, relative, or indexed file), it uses record I/O operations such as add, update, and delete record. The organization of the file determines the types of record operations permitted.

If the record accessing capabilities of RMS are not used, programs can access the file as an array of virtual blocks. To process a file at this level, programs use a type of access known as block I/O.

File Processing

At the file level, that is, independent of record processing, a program can:

- create a file
- open an existing file
- modify file attributes
- extend a file
- close the file
- delete a file

Once a program has opened a file for the first time, it has access to the unique internal ID for the file. If the program intends to open the file subsequently, it can use that internal ID to open the file and avoid any directory search.

Record I/O Processing

The organization of a file, defined when the file is created, determines the types of operations that the program can perform on records. Depending on file organization, RMS permits a program to perform the following record operations:

- Read a record. RMS returns an existing record within the file to the program.
- Write a record. RMS adds a new record that the program constructs to the file. The new record cannot replace an already existing record.
- Find a record. RMS locates an existing record in the file. It does not return the record to the program, but establishes a new current position in the file.
- Delete a record. RMS removes an existing record from the file. The delete record operation is not valid for the sequential file organization.
- Update a record. The program modifies the contents of a record in the file. RMS writes the modified record into the file, replacing the old record. The update record operation is not valid for sequential file organizations, except for sequentially organized disk files.

Sequential File Record I/O — In a sequential file organization, a program can read existing records from the file using sequential, RFA, or, if the file contains fixed-length records, random record access mode. New records can be added only to the end of the file and only through the use of sequential or random record access mode.

The find operation is supported in both sequential and RFA record access modes. In sequential record access mode the program can use a find operation to skip records. In RFA record access mode, the program can use the find operation to establish a random starting point in the file for sequential read operations.

The sequential file organization does not support the delete operation, since the structure of the file requires that records be adjacent in and across virtual blocks. A program can, however, update existing records in sequential disk files as long as the modification of a record does not alter its size.

Relative File Record I/O — Relative file organization permits programs greater flexibility in performing record operations than does sequential organization. A program can read existing records from the file using sequential, random, or RFA record access mode.

New records can be sequentially or randomly written as long as the intended record cell does not already contain a record. Similarly, any record access mode can be used to perform a find operation. After a record has been found or read, RMS permits the delete operation. Once a record has been deleted, the record cell is available for a new record. A program can also update records in the file. If the format of the records is variable, update operations can modify record length up to the maximum size specified when the file was created.

Indexed File Record I/O — Indexed file organization provides the greatest flexibility in performing record operations. A program can read existing records from the file in sequential, RFA, or random record access mode. When reading records in random record access mode, the program can choose one of four types of matches that RMS performs using the program-provided key value. The four types of matches are:

- exact key match
- approximate key match
- generic key match
- approximate and generic key match

Exact key match requires that the contents of the key in the record retrieved precisely match the key value specified in the program read operation.

The approximate match facility allows the program to select either of the following relationships between the key of the record retrieved and the key value specified by the program:

- equal to or greater than
- greater than

The advantage of this kind of match is that if the requested key value does not exist in any record of the file, RMS returns the record that contains the next higher key value. This allows the program to retrieve records without knowing an exact key value.

Generic key match means that the program need specify only an initial portion of the key value. RMS returns to the program the first occurrence of a record whose key contains a value beginning with those characters. This allows the program to retrieve a class of records, for example, all employee records in the personnel file with a name field beginning with M.

The final type of key match combines both generic and approximate facilities. The program specifies only an initial portion of the key value, as with generic match. Additionally, a program specifies that the key data field of the record retrieved must be either:

- equal to or greater than the program-supplied value

- greater than the program-supplied value

RMS also allows any number of new records to be written into an indexed file. It rejects a write operation only if the value contained in a key of the record violates a user-defined key characteristic (e.g., duplicate key values not permitted).

The find operation, similar to the read operation, can be performed in sequential, RFA, or random record access mode. When finding records in random record access mode, the program can specify any one of the four types of key matches provided for read operations.

In addition to read, write, and find operations, the program can delete any record in an indexed file and update any record. The only restriction RMS applies during an update operation is that the contents of the modified record must not violate any user-defined key characteristic (e.g., key values cannot change and duplicate key values are not permitted).

Block I/O Processing

Block I/O allows a program to bypass the record processing capabilities of RMS entirely. Rather than performing record operations through the use of supported record access modes, a program can process a file as a structure consisting solely of blocks.

Using block I/O, a program reads or writes blocks by identifying a starting virtual block number in the file and a transfer length. Regardless of the organization of the file, RMS accesses the identified block or blocks on behalf of the program.

Since RMS files, particularly relative and indexed files, contain internal information meaningful only to RMS itself, DIGITAL does not recommend that a file be modified by using block I/O. The presence of the block I/O facility, however, does permit user-created record formats on a Files-11 disk volume or ANS magnetic tape volume.

RMS RUN-TIME ENVIRONMENT

The environment within which a program processes RMS files at run time has two levels, the file processing level and the record processing level.

At the file processing level, RMS and the operating system provide an environment permitting concurrently executing programs to share access to the same file. RMS ascertains the amount of sharing permissible from information provided by the programs themselves. Additionally, at the file processing level, RMS provides facilities allowing programs to exercise as little or as much control over buffer space requirements for file processing as desired.

At the record processing level, RMS allows programs to access records in a file through one or more record access streams. Each record access stream represents an independent and simultaneously active series of record operations directed toward the file. Within each stream, programs can perform record operations synchronously or asynchronously. That is, RMS allows programs to choose between receiving control only after a record operation request has been satisfied (synchronous operation) or receiving control before the request has been satisfied (asynchronous operation).

For both synchronous and asynchronous record operations, RMS provides two record transfer modes, move mode and locate mode. Move mode causes RMS to copy a record to/from an I/O buffer from/to a program-provided location. Locate mode allows programs to process retrieved records directly in an I/O buffer.

Run-time File Processing

RMS allows executing programs to share files rather than requiring them to process files serially. The manner in which a file can be shared depends on the organization of the file. Program-provided information further establishes the degree of sharing of a particular file.

File Organization and Sharing — With the exception of magnetic tape files, which cannot be shared, an RMS file can be shared by any number of programs that are reading, but not writing, the file. Sequential disk files can be shared by multiple readers and multiple writers, but they are responsible for any record locking required to handle multiple readers and writers properly.

Program Sharing Information — A program specifies what kind of sharing actually occurs at run time. The user controls the sharing of a file through information the program provides RMS when it opens the file. First, a program must declare what operations (e.g., read, write, delete, update) it intends to perform on the file. Second, a program must specify whether other programs can read the file or both read and write the file concurrently with this program.

These two types of information allow RMS to determine if multiple user programs can access a file at the same time. Whenever a program's sharing information is compatible with the corresponding information another program provides, both programs can access the file concurrently.

Buffer Handling — To a program, record processing under RMS appears as the direct movement of records between a file and the program itself. Transparently to the program, however, RMS reads or writes the blocks of a file into or from internal memory areas known as I/O buffers. Records within these buffers are then made available to the program. Users can control the number and size of buffers. For sequential record access, users can choose an optional I/O read-ahead and write-behind buffer management. For magnetic tape file access, they can control the number of buffers for multiple buffering. For sequential disk files, users can specify the number of blocks that are to be transferred whenever RMS performs an I/O operation.

Run-time Record Processing

After opening a file, a program can access records in the file through the RMS record processing environment. This environment provides three facilities:

- record access streams
- synchronous or asynchronous record operations
- record transfer modes

Record Access Streams — In the record processing environment, a program accesses records in a file through a record access stream. A record access stream is a serial sequence of record operation requests. For example, a program can issue a read request for a particular record, receive the record from RMS, modify the contents of the record, and then issue an update request that causes RMS to write the record back into the file. The sequence of read and update record operation requests can then be performed for a different record, or other record operations can be performed, again in a serial fashion. Thus, within a record access stream, there is at most one record being processed at any time.

For relative and indexed files, RMS permits a program to establish multiple record access streams for record operations to the same file. The presence of such multiple record access streams allows programs to process in parallel more than one record of a file. Each stream represents an independent and concurrently active sequence of record operations.

As an example of multiple record access streams, a program could open an indexed file and establish two record access streams to the file. The program could use one record access stream to access records in the file in random access mode through the primary index. At the same time, the program could use the second record access stream to access records sequentially in the order specified by an alternate index.

Synchronous and Asynchronous Record Operations — Within each record access stream, a program can perform any record operation either synchronously or asynchronously. When a record operation is performed synchronously, RMS returns control to a program only after the record operation request has been satisfied (e.g., a record has been read and passed on to the program).

If the programming language allows asynchronous processing, RMS can return control to a program before the record operation request has been satisfied. A program can use the time required for the physical transfer to perform other computations. A program cannot, however, issue a second record operation through the same stream until the first record operation has completed. To ascertain when a record operation has actually been performed, a program can specify completion routines or issue a wait request and regain control when the record operation is complete.

Record Transfer Modes — A program can use either of two record transfer modes to gain access to each record in memory:

- move mode
- locate mode

Move mode means that the individual records are copied between the I/O buffer and a program. For read operations, RMS reads a block into an I/O buffer, finds the desired record within the buffer, and moves the record to a program-specified location in its work space. For write operations, the program builds or modifies a record in its own work space and RMS moves the record to an I/O buffer. RMS supports move mode record operations for all file organizations.

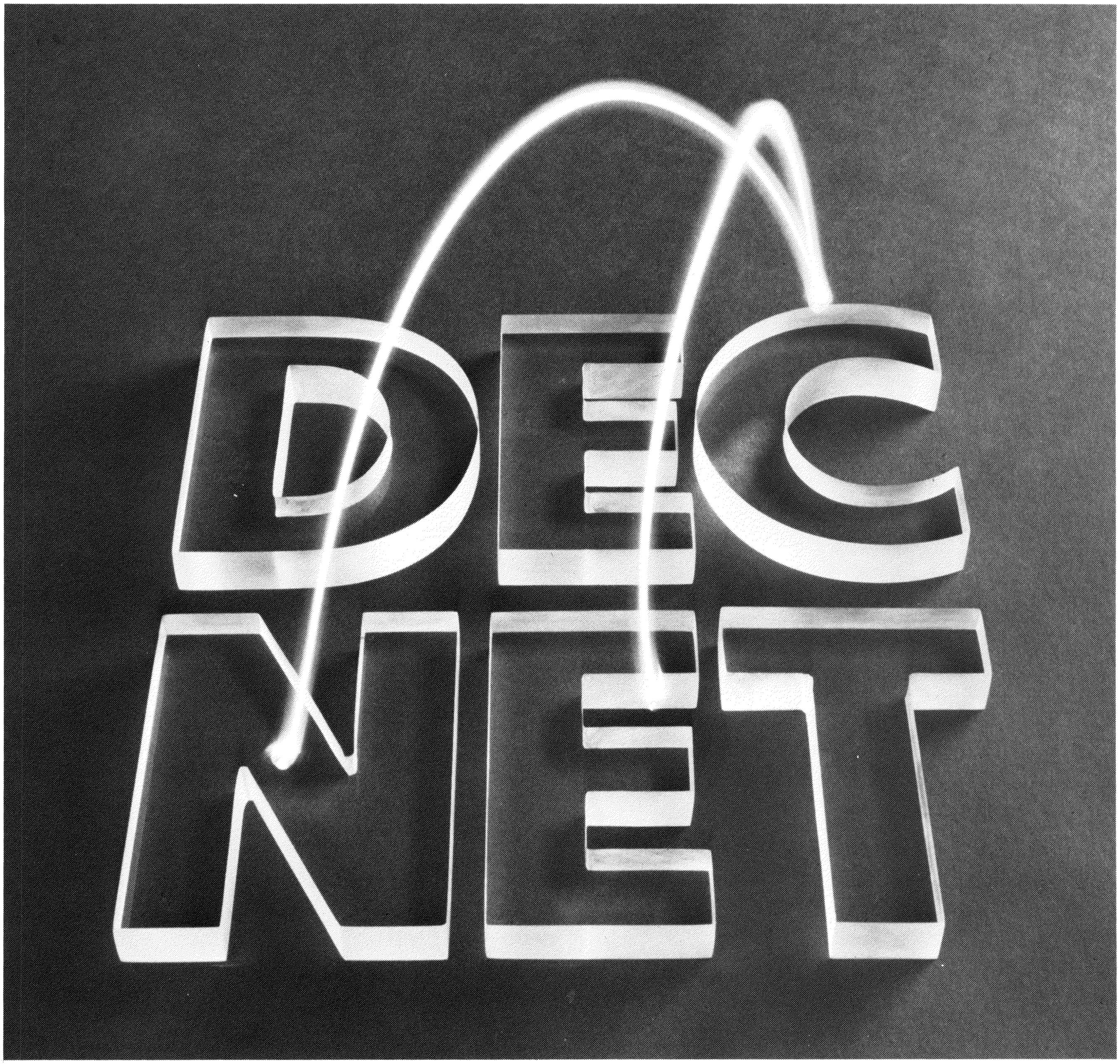
Locate mode enables programs to read records directly in an I/O buffer. Locate mode reduces the amount of data movement, thereby saving processing time. RMS provides the program with the address and size of the record in the I/O buffer. RMS supports locate mode record transfers on all file organizations for read operations only.

RMS Record Locking

VAX-11 RMS provides a record locking capability for files that use the relative and indexed organization. This provides control over operation when the file is being accessed simultaneously by more than one program and/or more than one stream in a program. Record locking makes certain that when a program is adding, deleting, or modifying a record on a given stream, another program or stream is not allowed access to the same record or record cell. There are two varieties of record locking and unlocking:

- Automatic Record Locking — The lock occurs on every execution of a $FIND or $GET macro instruction, and the lock is released when the next record is accessed, the current record is updated or deleted, the record stream is disconnected, or the file is closed. The $FREE macro instruction explicitly unlocks all records previously locked for a particular record stream. The $RELEASE macro instruction explicitly unlocks a specified record in a record stream.
- Manual Record Locking — In manual record locking, varying degrees of locking may be specified by setting bits in the record processing options field (ROP) of the RAB. The ULK bit specifies manual (as opposed to automatic) locking and unlocking. This bit specifies that locking will occur on the execution of a $GET, $FIND, or $PUT macro instruction and that unlocking may take place explicitly only via a $FREE or $RELEASE macro instruction. The NLK bit specifies that the record accessed with either a $GET or $FIND macro instruction is not to be locked, while the RLK specifies that a record may be accessible for read purposes but may not otherwise be accessed.

9 The Network Services



DECnet is a family of network products developed by Digital Equipment Corporation that adds networking capability to DIGITAL's computer families and operating systems. Using DECnet, various kinds of computer system networks can be constructed to facilitate remote communications, resource sharing, and distributed computation. DECnet is highly modular and flexible. It can be viewed as a set of tools or services from which a user selects those appropriate to build a network to satisfy the requirements of a particular application.

DIGITAL Network Architecture (DNA) provides the common network structure upon which all DECnet products are built. The architecture is designed to handle a broad range of application requirements because all the functions of the network from the user interface to physical link control are completely modular. DNA allows nodes to operate as switches, front-ends, terminal concentrators or hosts.

DECnet/VAX:

- provides an interprocess communication facility that is highly transparent and easy to use
- provides a high-level language programming interface
- allows programs to access files at other systems
- allows users and programs to transfer files between systems
- allows users to transmit command files to be executed in other systems
- allows an operator to down-line load RSX-11S system images into other systems

INTRODUCTION

With DECnet, a variety of computer networks can be implemented. They typically fall into one of three classes:

- Communications Networks: These networks exist to move data from one, often distant, physical location to another. The data may be file-oriented (as is often the case for remote job entry systems) or record-oriented (as occurs with the concentration of interactive terminal data). Interfaces to common carriers, using both switched and leased-line facilities, are normally a part of such networks. Such networks are often characterized by the concentration of all user applications programs and data bases on one or two large host systems in the network. Figure 9-1 illustrates such a network.

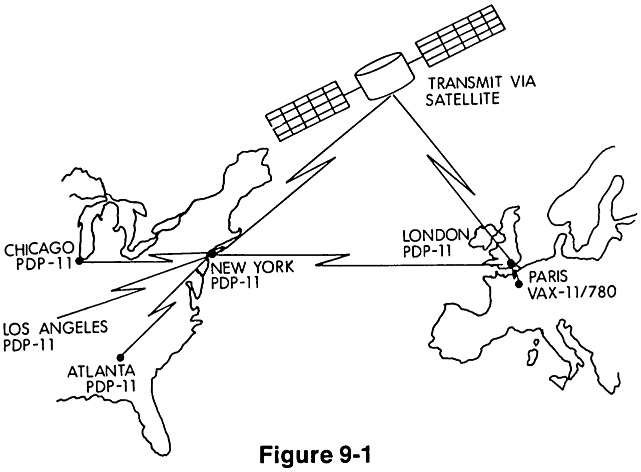


Figure 9-1
Communications Network

- Resource-Sharing Networks: These networks exist to permit the sharing of expensive computer resources among several computer systems. Shared resources not only include peripherals such as mass storage devices, they can also include logical entities, such as a centralized data base which is made available to other systems in the network. Such networks are often characterized by the concentration of high-performance peripherals, extensive data bases, and large programs on one or two host systems in the network, while the satellite systems have less expensive peripherals and smaller programs. Figure 9-2 illustrates a resource-sharing network.

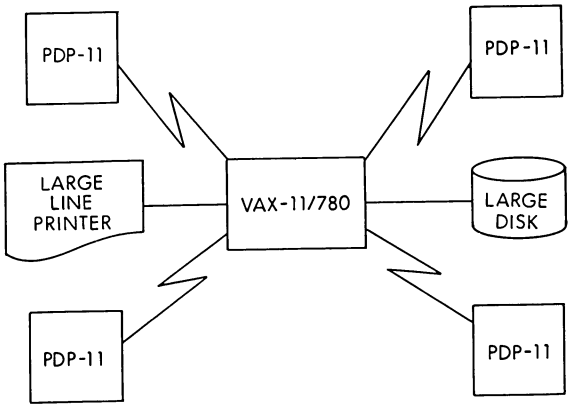


Figure 9-2
Resource-Sharing Network

- Distributed Computing Networks: These networks coordinate the activities of several independent computing systems and exchange data between them. Networks of this nature may have specific geometries (star, ring, hierarchy), but often have no regular arrangement of links and nodes. Such networks are usually configured so that the resources of a system are close to the users of those resources. Distributed computing networks are usually characterized by multiple computers with applications programs and data bases distributed throughout the network. Figure 9-3 and 9-4 illustrate two such networks.

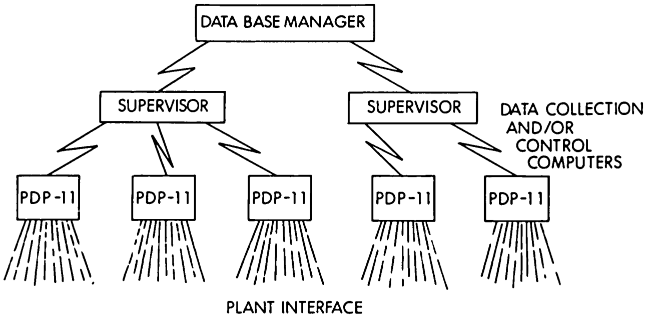


Figure 9-3
Typical Manufacturing Network

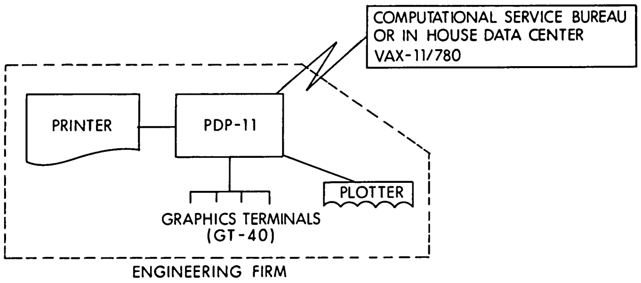

COMPUTATIONAL SERVICE BUREAU OR IN HOUSE DATA CENTER VAX-11/780
PRINTER
PDP-11
PLOTTER
GRAPHICS TERMINALS (GT-40)
ENGINEERING FIRM

Figure 9-4
Computational Network

When DECnet is used to connect heterogeneous systems, each node of the network has both common DECnet attributes and system-specific attributes. The attributes provided by DECnet/VAX include:

- Interprocess (Task-to-Task) Communication: Programs executing on one system can converse with programs executing on other systems.
- Inter-System File Transfer: A program or a command language user can transfer an entire data file from one system to another.
- Inter-System Resource Sharing: Programs executing on one system can access files physically located at other systems in the network. Access to devices in other systems is provided only through the file system of the target node and is subject to that system's DAP implementation or file system restrictions.

- Down-Line System Loading: Initial load images for RSX-11S systems in the network can be stored on the host VAX-11/780 system, and loaded on request into PDP-11 systems configured for the RSX-11S operating system.
- Down-Line Command File Loading: Programs or command language users can send command files to a remote node to be executed there. However, no status information or error messages are returned.
- High-Level Language Interface: This facility allows programs written in any VAX-11 native programming language to access some of the network facilities.

When a VAX/VMS system operates as a node in a DECnet configuration, the types of functions it can perform will be somewhat dependent on other DECnet products in the configuration. This is because two nodes communicating with each other will always be restricted to those functions which are common to both of them. Table 9-1 lists the various DECnet systems which may be configured with DECnet/VAX and the functions each may perform.

Table 9-1
DECnet Products and Functions

	DECnet-11M Version 2.0	DECnet-11S Version 2.0	DECnet-11D Version 2.0	DECnet-IAS Version 2.0	DECnet/E Version 1.0	DECnet-RT Version 1.0	DECnet-VAX Version 1.0	DECnet-20 Version 1.0
Task-to-Task	YES	YES	YES	YES	YES	YES	YES	YES
Intersystem File Transfer	YES	NO	YES	YES	YES	YES	YES	NO
Command/Batch File Submission	YES[1]	NO	YES[1]	YES[1]	YES	YES	YES[1]	NO
Command/Batch File Execution	YES	NO	YES	YES	YES	NO	YES	NO
Remote File Access	YES	YES[2]	YES	YES	NO	YES	YES	NO
Down-Line System Loading	YES	NO	YES	YES	NO	NO	YES	NO
Down-Line Task Loading	YES	NO	YES	YES	NO	NO	NO	NO

[1] Cannot submit files to DECnet/E systems. Can tell DECnet/E to execute batch files already at the DECnet node.

[2] Offers local users network access to remote file systems. Does not allow users on remote systems to access local files.

DIGITAL NETWORK ARCHITECTURE

The DIGITAL Network Architecture (DNA) is a set of protocols (rules) governing the format, control, and sequencing of message exchange for all DECnet implementations. DNA controls all data that travels throughout a DECnet network and provides a modular design for DECnet.

Its functional components are defined within four distinct layers: User Layer, Logical Link Control Layer, Physical Link Control Layer, and Hardware Layer. Each layer performs a well-defined set of network functions (via network protocols) and presents a level of abstraction and capability to the layer above it.

User Layer: At the user layer, a message can be generated to be transmitted to another process. The message can be generated at a terminal (such as a request to transfer a file), or in a process (using RMS or System Services). It is this layer that is "visible" to the user. All other layers are transparent, thus allowing network use without the necessity of learning the many protocols required to transmit data between processes.

Logical Link Control Layer: Processes access the network at the logical link layer. The Network Services protocol (NSP) accepts the data from the user layer, and formats the data for multiplexed transmission by the communications hardware. NSP also accepts incoming data from remote processes, separates the data into logical link streams, and delivers the data to the user process.

Physical Link Control Layer: DIGITAL Data Communications Message Protocol (DDCMP) controls physical links. Like other line protocols, DDCMP maintains control over a physical communications link to provide an error-free, sequential data path over a generally error-prone medium. DDCMP accomplishes this with a Cyclic Redundancy Check (CRC) for error detection, transmission for error corrections, and numbered data messages to ensure sequential transfer of data.

Hardware Layer: The hardware layer transmits and receives data over the physical link. Device drivers or interrupt service routines exchange data with the interface and handle synchronization and modem control for the hardware.

PROGRAMMING AND OPERATING INTERFACE

The following paragraphs describe the ways in which programmers, operators, and terminal users can interact with DECnet. Essentially, there are three ways in which users can employ DECnet features:

- Interprocess or "task-to-task" communication. The DECnet/VAX programming interface allows both transparent and non-transparent interprocess communication.
- File and record operations. DECnet/VAX allows file and device access to programs and terminal users. A terminal user can create, transfer, and delete files in other nodes. Programs also have this capability, with the added ability to read and write records. These capabilities provide the tools for creation of distributed data bases.
- Down-line loading. DECnet/VAX can down-line load RSX-11S system images, provided the target system includes a ROM bootstrap. No peripheral device for loading, such as floppy disk or paper tape, is required for such nodes.

INTERPROCESS COMMUNICATION

Interprocess communication is the one feature common to all DECnet implementations. Interprocess communication allows programs (tasks, processes, etc.) to create one or more logical links, which are full-duplex virtual data paths. Programs have the capability to create these connections, transmit and receive data over them, and destroy them. Data transmissions can be done on a normal or priority (i.e., interrupt) basis.

In DECnet/VAX, the interface itself can be transparent (each program looks like a sequential device to the other), or non-transparent (each program knows that it is using DECnet and has the opportunity to acquire information about the network). Each access method has its advantages. Transparent access is easier to learn, and it allows great flexibility in that the location of files, devices, and the program itself need not be determined until run time. Under non-transparent access, the programmer can take advantage of known properties of the protocols, providing the ability to transmit and receive interrupt messages, connect initiates, and disconnect notification.

Transparent Interprocess Communication

In transparent access, the program opens the network interchange as if it were preparing device access, and then performs a series of reads and writes, just as it would to a pair of serial devices, one for input (reception) and the other for output (transmission). By its very nature, transparent access has no calls specifically associated with DECnet. The calls used for interprocess communication are the same as the calls used for accessing a sequential file in a high-level language: OPEN, CLOSE, READ, WRITE, etc. The programmer can choose to include the target node name in the OPEN statement, or can defer assignment using logical names.

Non-transparent Interprocess Communication

In non-transparent access, a program can obtain information about the network status to control the nature of its communication with other processes or tasks. Non-transparent access is available only through calls to operating system service procedures. A program can issue the following requests:

- CONNECT — Establish a logical link (the analog of OPEN).
- CONNECT REJECT — Reject a connect initiate.
- RECEIVE — Receive a message (the analog of GET).
- SEND — Transmit a message (the analog of PUT).
- SEND INTERRUPT MESSAGE — Transmit a high-priority message.
- DISCONNECT — Terminate a conversation (the analog of CLOSE).

The process can send optional data along with the connect request, for example, the size or number of messages that it wants to send. The receiving process or task can accept or reject the connect initiate. A process can accept multiple connect requests.

A process can send or receive unsolicited messages to or from another process or task. Unsolicited message traffic is essentially no different from solicited message traffic except that it uses a system's software interrupt mechanism to transmit a message. A logical link therefore has two subchannels over which messages can be transmitted: one for normal messages and another for high-priority messages. In DECnet/VAX, an interrupt message is written to a mailbox that a process supplies for that purpose. The process can request that an asynchronous system trap routine be executed when a message is queued to that mailbox.

A program can issue a synchronous disconnect, which guarantees the receiver that it got every message that was sent, or it can issue a disconnect abort, which terminates the logical link immediately.

In DECnet/VAX, a program using non-transparent access normally opens a control path directly to the Networks Ancillary Control Process (NETACP), and designates one or more mailboxes for receiving information from the NETACP about the logical or or physical links over which the process is communicating. The NETACP can notify a process when:

- A partner requests a synchronous disconnect.
- A partner requests a disconnect abort.
- A partner exits.
- A physical link goes down.
- An NSP protocol error is detected.

If the process has the diagnostic privilege, it can also stop and start DDCMP protocol over a physical link.

DATA TRANSFER OPERATIONS

Using the VAX/VMS command language, operators and terminal users can:

- copy files from the VAX-11 system to another system
- delete files in other systems
- transfer command files for executing on other systems

Programs have access to all the above file operations, and they can also read and write records sequentially and randomly.

Command Language

DECnet/VAX-11 supports the following VAX/VMS commands:

$ APPEND
$ ASSIGN
$ COPY
$ DEASSIGN
$ DEFINE
$ DELETE
$ SUBMIT
$ TYPE

The following examples illustrate the $ COPY and $ SUBMIT commands:

$ COPY BOSTON::DBA1:TEST.DAT DENVER::DMA2:

transfers a file named TEST.DAT from the disk (DBA1:) at the node named BOSTON to the disk (DMA2:) at the node named DENVER.

Using the VAX/VMS command SUBMIT, a terminal user can send a command file to another system in a network, and have it executed at that node. For example, the command:

$ SUBMIT/REMOTE:WASHDC INITIAL.COM

transfers the command file named INITIAL.COM from the host system to the node named WASHDC, where the command file is executed. Command files must be written in the command language of the target system. No status information or messages are returned to the sender.

Programmed Data Transfer

For more complicated operations on remote files, DECnet supports the following RMS macros for both sequential or relative file access:

- $CREATE — Create and open a file.
- $ERASE — Delete a closed file.
- $OPEN — Open a file.
- $CLOSE — Close an open file.
- $CONNECT — Establish a record processing stream.
- $DISCONNECT — Terminate a record processing stream.
- $GET — Read a record.
- $PUT — Write a record.
- $UPDATE — Modify an existing record.
- $DELETE — Delete a record from a relatively organized file.
- $READ — Read a block.
- $WRITE — Write a block.
- $FIND — Position to a specified record.
- $REWIND — Position to the beginning of a file.
- $SPACE — Position to the next block.

Figure 9-6, showing a MACRO program transferring a sequential file from one device to another, illustrates RMS file transfer under DECnet.

```
            .TITLE DEMO1 - RMS FILE TRANSFER EXAMPLE
;
;       This program transfers a sequential file with variable length
;       records from one device to another. The devices are specified
;       by the logical names SRC and DST. For example, to display file
;       INVENTORY.DAT at node ALBANY on the line printer at node
;       BOSTON, execute the following procedure:
;
;       $ DEFINE SRC ALBANY::DBB3[XYZCO.STOCK]INVENTORY.DAT
;       $ DEFINE DST BOSTON::LPA0:
;       $DEMO1
;
            .SBTTL CONTROL BLOCK AND BUFFER STORAGE
            .PSECT      IMPURE NOEXE.LONG
;
;           Define the source file FAB and RAB control blocks.
;
SRC_FAB:
            $FAB        FAC=<GET>-
                        FOP=SQO-
                        FNA=SRC_NAM-
                        FNS=SRC_NAM-SIZ
SRC_RAB:
            $RAB        FAB=SRC_FAB-
                        RAC=SEQ-
                        UHF=BUFFER-
                        USZ=BUFFER_SIZ
;
;           Define the destination file FAB and RAB control blocks.
;
DST_FAB:
            $FAB        FAC=<PUT>-
                        FOP=SQO-
                        FNA=DST_NAM-
                        FNS=DST_NAM_SIZ-
                        ORG=SEQ-
                        RFM=VAR-
                        RAT=CR
DST_RAB:
            $RAB        FAB=DST_FAB-
                        RAC=SEQ
;
;           Define logical names for the source and destination files.
;
SRC_NAM:
            .ASCII      /SRC/
            SCR_NAM_SIZ==.-SRC_NAM
DST_NAM:
            .ASCII      /DST/
            DST_NAM_SIZ==.-DST_NAM
;
;           Allocate buffer space to be size of largest record.
;
BUFFER:
            BLKB        132
            BUFFER_SIZ=.-BUFFER

            .SBTTL      MAINLINE
            .PSECT      CODE NONRT,NYTB

            .ENTRY      DEMO1, ↑M<>
;
;           Put FAB and RAB addresses in registers for efficiency.
;
            MOVAB       W↑SRC_FAB,R6
            MOVAB       W↑SRC_RAB,R7
            MOVAB       W↑DST_FAB,R8
            MOVAB       W↑DST_RAB,R9
;
;           Open the SRL and DST files.
;
            $OPEN       FAB=R6
            BLBC        R0,30$
            $CONNECT    RAB=R7
            BLBC        R0,30$
            $CREATE     FAB=R8
            BLBC        R0,30$
            $CONNECT    RAB=R9
            BLBC        R0,30$
;
;           Transfer records until end-of-file is encountered.
;
10$;        $GET        RAB=R7
            CMPW        R0,"<RMS$_EOF&↑XFFFF>
            BEQL        20$
            MOVL        RAB$L_UBF(R7),RAB$L_RBF(R9)
            MOVW        RAB$W_RSZ(R7),RAB$↑_RSZ(R9)
            $PUT        RAB=R9
            BLBC        R0,30$
            BRB         10$
;
;           Close the SRC and DST files.
;
;     Note: in this example, the $DISCONNECT calls below are not
;     necessary because $CLOSE performs an implied
;     $DISCONNECT. They are included for symmetry.
;
20$:        $DISCONNECT RAB=R9
            BLBC        R0,30$
            $CLOSE      FAB=R8
            BLBC        R0,30$
            $DISCONNECT RAB=R7
            BLBC        R0,30$
            $CLOSE      FAB=R6
;
;           Exit to VMS. Also, enter here on detection of an error.
;
30$:        $EXIT_$R0       ; R0 = RMS completion code to
                            ; display on error condition
             END        DEMO1
```

Figure 9-6
RMS File Transfer Example

FORTRAN IV-PLUS Support

DECnet/VAX supports the following FORTRAN IV-PLUS statements:

- OPEN — Creates a file and connects it to a logical unit, or connects an existing file.
- CLOSE — Disconnects a file from a logical unit.
- DEFINE FILE — Establishes the size and structure of a file upon which direct access I/O is to be performed.
- READ — Performs formatted or unformatted input on a specified logical unit.
- WRITE — Performs formatted or unformatted output on a specified logical unit.
- ACCEPT — Performs formatted input on an implicit logical unit, normally with a terminal keyboard.
- TYPE — Performs formatted output on an implicit logical unit, normally with a terminal printer or display screen.
- PRINT — Performs formatted output on an implicit logical unit, normally with a line printer.
- REWIND — Repositions an open file to the beginning of the file.
- BACKSPACE — Repositions an open file to the beginning of the previous record.
- ENDFILE — Writes an end-of-file record on an open sequential file.
- FIND — Positions an open direct-access file to a record and sets the variable of the file to that record number.

The following example shows how a FORTRAN IV-PLUS program could retrieve records from a file at a node DALLAS and print them on the line printer at node BOSTON. If node names are omitted from the file specifications, the same routine prints a local file on a local line printer.

```
      CHARACTER*133 REC
      OPEN (UNIT=1,NAME='DALLAS::DBA1:FILE.DAT',TYPE='OLD')
      OPEN (UNIT=2,NAME='BOSTON::LPA1:')
10    READ (1,100,END=300)REC
100   FORMAT (A)
      WRITE (2,100)REC
      GO TO 10
300   CLOSE (UNIT=1)
      CLOSE (UNIT=2)
      END
```

FORTRAN IV-PLUS INTERTASK COMMUNICATION

There are three major steps in FORTRAN IV-PLUS intertask communication:

- Create a logical link between tasks.
- Send and receive messages (each message can be 1 to 512 bytes in length).
- Destroy the link at the end of the message dialog.

Creating a Logical Link Between Processes

The establishment of a logical link between processes requires "cooperation" between the two processes. That is, one process (the source process) must request that a logical link be created, and the other process (the target process) must agree to accept the request.

The source task issues a logical link connect request by including a task specifier in the source task's Open statement. This is illustrated in the following example:

```
OPEN (UNIT=7, NAME='DENVER::"TASK=ACC",ERR=200)
```

Here the NAME argument requests a logical link connection to target task ACC on node DENVER. The local node passes the logical link connect request to the remote node (using DECnet/VAX services). The remote node creates a process for the target task, and places the source task identifier in the logical name table under the name SYS$NET.

The target task identifies the source task requesting the logical link connect request by specifying SYS$NET as the NAME argument in the OPEN statement.

An example of the target task OPEN statement would be:

```
OPEN (UNIT=2,NAME=SYS$NET,ERR=700)
```

Sending and Receiving Messages

After the logical link has been created, the tasks must "cooperate" with each other. That is, for each message sent by a task (WRITE statement), the receiving task must issue a corresponding receive (READ statement).

In addition, the tasks must ensure that enough buffer space is allocated for messages, that the end of dialog can be determined, and which of the tasks will disconnect the logical link (CLOSE statement).

Disconnecting the Logical Link

Either task can disconnect the logical link by calling CLOSE. CLOSE aborts all pending sends and receives, disconnects the link immediately, and frees the channel number associated with the logical link.

MACRO NON-TRANSPARENT INTERTASK COMMUNICATION

The VAX-11 MACRO language permits the user to perform non-transparent intertask communication as well as transparent communication, allowing greater flexibility and control in network operations. Non-transparent intertask communication includes the following capabilities:

- Associate a mailbox with the I/O channel (over which the logical link will be created). The mailbox can then receive unsolicited messages sent by a remote task, or notifications affecting the status of the logical link. For example, the remote task accepted or rejected a connect request, or the cooperating task disconnected or destroyed the link.
- A task can declare itself as a network task to accept multiple logical link connect requests.
- A source task can send a logical link connect request to the target task. The source task can optionally send 1 to 16 bytes of data to the target task at the same time it issues the connect request.
- The target task can accept or request the connect request. It can send 1 to 16 bytes of optional data back to the source task at the same time it accepts or rejects the connect request.
- A non-transparent task can also accept or reject connect requests received from tasks written using transparent intertask communication system service calls.
- Either task can send or receive a 1- to 16-byte interrupt message after the logical link is created.
- Either task can abort the link immediately, or issue a synchronous disconnect. The task disconnecting or aborting the logical link can also send 1 to 16 bytes of optional data to the remote task at the same time it disconnects or aborts a logical link.

Creating a logical link in non-transparent intertask communication requires that both the source and target tasks call the $ASSIGN system service in order to:

- Assign to device NETO:
- Request a channel number for the logical link.
- Associate a previously created mailbox with the channel.

In addition to $ASSIGN, the following VAX/VMS system services can be used in non-transparent intertask communication. These are:

- $QIO(IO$_ACCESS) — Request a Logical Link Connection
- $QIO(IO$_ACCESS) — Accept a Logical Link Connection Request
- $QIO(IO$_ACCESS!IO$M_ABORT) — Reject A Logical Link Connection Request
- $QIO(IO$_WRITEVBLK) — Send a Message to a Remote Task
- $QIO(IO$_READVBLK) — Receive a Message from a Remote Task
- $INPUT — Read a Message
- $QIO(IO_WRITEVBLK!IO$M_INTERRUPT) — Send an Interrupt Message to a Remote Task
- $QIO(IO_DEACCESS!IO$M_SYNCH) — Synchronously Disconnect the Logical Link
- $QIO(IO_DEACCESS!IO$M_ABORT) — Abort a Logical Link
- $QIO(IO$_ACPCONTROL) — Declare a Network Name
- $DASSGN — Disconnect the Logical Link

DOWN-LINE LOADING AND NETWORK STATUS

This interface for the operator at the host VAX-11/780 system is a function of the NCP utility that accepts parameters such as the location and name of the RSX-11S system image and the name of the node to be sent the load image. For example:

```
$ NCP
#1> LOAD NODE DALLAS
#2> LOAD NODE BOSTON FROM SYSMON.EXE
```

In the first request, NCP starts down-line loading the default operating system image to node DALLAS, while in the second request, NCP starts loading a specific system image.

Booting the RSX-11S operating system requires the presence of a read-only memory bootstrap program and, for cold start, an operator to power-on the system and switch the DMC11 to "remote load detect." For warm start, when an operating system is already running, no operator intervention is required, as the system automatically jumps to the bootstrap.

For cold start, the operator of the RSX-11S system starts the ROM bootstrap. It sends a special BOOT-ME message to the host, which automatically sends the proper system over the link.

For warm start, the VAX-11/780 host operator may decide to load a different system in the satellite machine. In such cases, the utility will send over the appropriate BOOT messages, and the satellite will automatically stop any current processing and go into BOOT mode. No operator is required at the RSX-11S satellite in this case.

Finally, a watchdog timer is available that will automatically put the RSX-11S system into BOOT mode whenever the operating system fails to reset a bit within a specified period. In this case, no operator intervention is required at either end. In all cases, the satellite system can start automatically after a boot operation.

NCP also enables the operator at the VAX/VMS system to:

- SHOW PATHS — Identify the physical links connected to the system.
- SHOW COUNTS — Display the current network throughput and error statistics.
- ZERO COUNTS — Initialize the statistics counters.
- SHOW STATUS — Display the status of a node in the network or a particular physical link on any node. Node status includes: ON (allow logical links), SHUT (active, but allow no new logical links), MAINTENANCE (no logical links, but physical links may be active; for example, down-line load in progress), and OFF. Line status includes ON, MAINTENANCE, and OFF.
- SET STATE — Set the status for each physical link and for the local system.

- LOOP — Perform loop-back tests with a particular node or with a particular physical link in the local system. The operator can, for example, test the local NSP functions, the local modem, the remote modem, or the remote NICE functions.

BLANK

10 The Support Services

DIGITAL offers special services to help before, during and after system installation. DIGITAL's sales force is the primary contact for all products and services.

First, with your sales representative, you study your application and requirements. During this time, you work closely with the salesperson to determine your computing needs. The sales representative may call in software and hardware specialists to help answer specific questions. These specialists are trained to design systems using DIGITAL's standard products, and if necessary, to suggest alternate solutions for special situations.

Once you have determined the exact nature of your requirements, your sales representative helps you select a system configuration. You review the site requirements outlined in the *Site Preparation Guide* (assuring adequate floor space, electrical capacity, air conditioning and humidity control, etc.). You select a Field Service maintenance plan that is appropriate to your needs and budget. Your salesperson writes up your order and helps arrange financing.

While waiting for delivery of your system, you can train your personnel by taking advantage of DIGITAL's extensive educational programs. By purchasing a system, you obtain training credits which can be applied to the cost of some of DIGITAL's courses.

On delivery, DIGITAL's Hardware Field Service and Software Support organizations are on hand to ensure smooth installation. Specialists install the hardware and the software and run extensive tests to ensure that the system is correctly installed and performing properly.

Following installation, DIGITAL's support organizations are available to help with any special needs that may arise both during and after the warranty period. The hardware Field Service organization includes representatives in locations throughout the world who can provide preventive and remedial maintenance to keep your system hardware performing optimally. Software specialists, trained and proficient in the use of your software system, can be called in to help with any questions that arise in its operation or use. In addition, software consultants and the Computer Special Systems group may be contracted to help design and build systems to meet individual requirements.

INSTALLATION

Upon delivery of your system, your Field Service account representative will schedule installation of the hardware components. During installation, Field Service engineers supervise the uncrating and placement of equipment, cable connections, and powering-up of components. They test the hardware by running a system exerciser — a complete diagnostic package. Once hardware reliability is confirmed, the Field Service engineers coordinate with software support personnel to install and test the operating system. They use the User Environmental Test Package to exercise the system software components. This package runs customer-type routines (compilations and assemblies) and serves as both a final test of the system and as a demonstration of system operation.

Finally, Field Service completes DIGITAL forms certifying successful installation. You acknowledge system acceptance by signing the Field Service Labor Accounting and Reporting System form.

HARDWARE SERVICES

On-site warranty begins 30 days after the system was shipped or upon customer acceptance, whichever occurs first. Hardware components in a packaged system are covered under warranty for 90 days. OEMs must buy back a minimum installation and 30-day on-site warranty; an optional installation and 90-day on-site warranty is available. During the warranty period, DIGITAL will perform remedial maintenance on any defective components.

Once the warranty period ceases, your maintenance plan becomes effective.

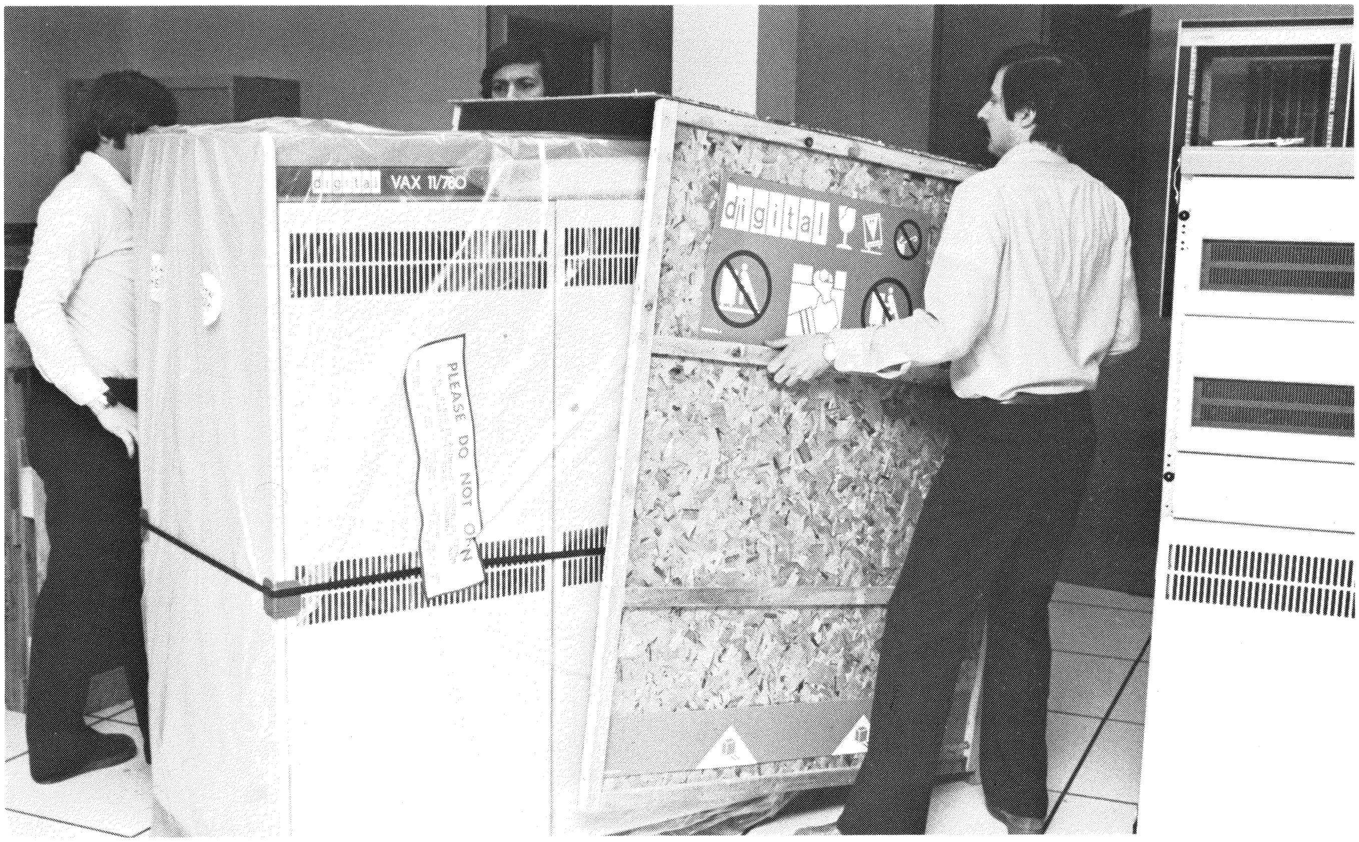
Most customers select a DECservice Agreement. The DECservice Agreement is for customers who require high availability, whose system is critical for daily operation. This plan includes all materials and labor, preventive maintenance, installation of engineering changes, and the use of licensed diagnostics. It guarantees 4-hour response on service calls and continuous service until repairs are complete.

If your availability requirements are not stringent, you may choose a Basic Agreement. The Basic Agreement includes all materials and labor, installation of engineering changes, and the use of licensed diagnostics. However, preventive maintenance occurs less frequently than under the DECservice Agreement.

With either of these service agreements, Remote Diagnosis is available. Remote diagnosis service capabilities are normally installed and used by Field Service during the warranty period. Remote Diagnosis allows a DIGITAL Field Service engineer, with your permission, to access your system from a remote site for the purpose of exercising diagnostics to determine hardware problems. Its main benefit is increased system availability, due to shortened diagnosis and repair time. Under both the DECservice Agreement and the Basic Agreement, you may obtain coverage from 8 hours a day 5 days a week, up to 24 hours a day 7 days a week. Your sales representative is trained to help you select the best coverage.

Two non-contract maintenance options are also available: Per-Call and Self-Maintenance.

Per-Call service is provided for customers who do not need immediate response or continuous effort. Generally



per-call arrangements are useful for installations that are non-critical, have redundant backup equipment, or are self-maintained. Per-call service is also offered as a supplementary service (on a "best-efforts" basis) for service agreement customers, if remedial service is required outside the hours of contract coverage.

Off-Site Services are available for customers who prefer to maintain their own equipment. DIGITAL offers many kinds of assistance, including off-site repair of major equipment and modules; recommended spares service, spare parts, tool kits, and test equipment; maintenance documentation and engineering updates; emergency parts service; and hardware maintenance training. A license is required to use DIGITAL diagnostics when no service agreement is in effect.

If you have a service agreement, you need only contact Field Service by dialing the phone number provided during installation whenever you require Field Service assistance to solve a hardware-related problem. If you do not have a service agreement, you can still obtain help. Call your local Field Service branch office for details.

SOFTWARE SERVICES

Software Services is committed to maintaining a high level of support for the VAX-11/780 software. Software specialists have been specially trained in VAX-11/780 software in order to provide the expert knowledge and experience necessary to analyze your needs, to identify those DIGITAL services that will help you meet those needs, and to deliver those services through your local software specialist. In addition, your local software specialist has access to backup support at the regional and corporate levels, when necessary. This means that DIGITAL's total software resources and expertise are available to support your VAX-11/780 software product.

Software Warranty

The VAX-11/780 software package is covered by DIGITAL's Software Warranty. This warranty consists of installation of the software, 90 days of remedial support from your local DIGITAL office, and one year of software maintenance service.

During the 90 days of remedial service, if you encounter a problem that DIGITAL determines to be a defect in the current unaltered release of the software, DIGITAL will provide the following remedial services, on site where necessary:

- If the software is inoperable, apply a temporary correction or make a reasonable attempt to develop an emergency by-pass.
- Help you prepare and submit a Software Performance Report.

During the one year of software maintenance services, the following service components are in effect:

- Software Dispatch — This monthly publication is automatically distributed and contains information on reported software problems and their solutions, general software information, system enhancements, and new software products.
- Software Updates — For a media charge only, you can receive the most recent version of the existing software containing the most recent fixes and new features. You are informed through the Software Dispatch before a new release is delivered so that you can prepare your operation.
- Software Performance Report (SPR) Service — SPRs are a formal problem-reporting service which you may use to document any detected software problems. You are assured a response to every SPR submitted to DIGITAL. SPRs are used by DIGITAL as a basis for software remedy and enhancement. SPRs of a general nature are printed in the Software Dispatch.

Post-Warranty

Once the warranty period expires, you may elect to continue maintenance services by purchasing a yearly subscription to the Binary Program Update Service. This service is a continuation of the three maintenance services you receive under the software warranty, namely:

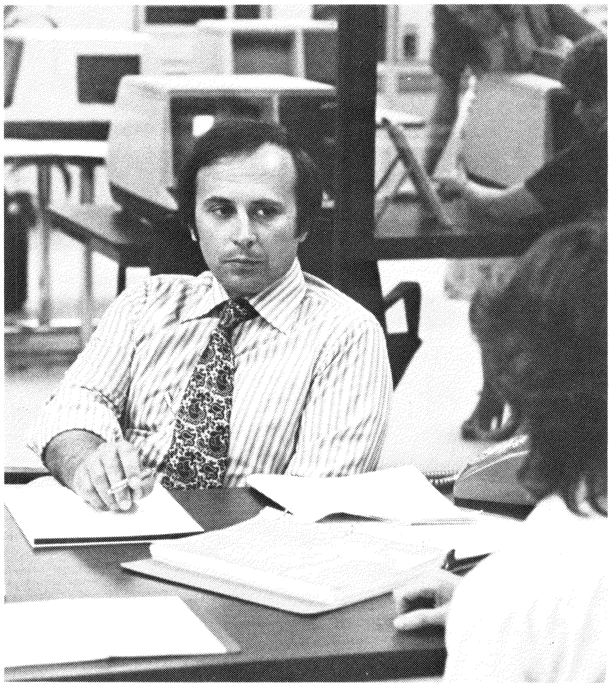
- Software Dispatch
- Software Update
- Software Performance Reports

Appropriate order forms will be supplied automatically when your warranty is about to expire.

Consulting Services

If you need expert software assistance, DIGITAL provides software consultants. Software consultants are professionals trained in DIGITAL products. They are experienced in designing and writing custom software as well as tailoring DIGITAL software to meet specific needs. Their expertise can be applied to any phase of any application. They are specially trained to help resolve compatibility and conversion problems with your application. They can provide operational and administrative services, such as developing an operations manual or providing on-site training for your staff. They can perform system analysis after your system has been in operation for some months.

Consulting services are available in a variety of plans:



- Resident Consulting — for those customers who need a full-time resident specialist. Residents are particularly useful in new, complex installations, or in critical, long-term projects. Residents are usually on site for 6-month or 12-month periods. Arrangements can be made to adjust the length of service to suit your needs (with a minimum of 6 months required).
- Weekly Consulting — for customers with definite, but short-term consulting needs. Weekly consulting includes 40 hours of on-site support. Consecutive weeks of service may be ordered as needed.
- Per-Call Consulting — for customers with irregular or infrequent consulting needs. Per-call (hourly) services may be ordered as needed and generally extend from a few hours to a few days.

To request consulting services, or for more information, contact your local DIGITAL office.

EDUCATIONAL SERVICES

To train your personnel before, during, and after installation, DIGITAL provides comprehensive educational programs. Trained instructors give standard courses in system management, operations, hardware, and software at DIGITAL's worldwide training centers. In addition, special on-site training and custom courses can be arranged.

Courses fall into four general categories:

- General Interest Computer Science Courses — These are not geared to a specific DIGITAL system or product, but provide a technical foundation for personnel who have little previous computer experience.
- Software Systems Courses — These courses are designed to train users, programmers, and operators in the efficient and knowledgeable use of DIGITAL's operating systems, languages, and utilities. Courses are available for both beginning and advanced users. For the most part, these courses assume that the student has general computer and programming knowledge.
- Hardware Courses — These are designed for customers who intend to service their own equipment or who simply want a general understanding of the components in their system. Courses are available in general hardware familiarization, hardware troubleshooting, and hardware maintenance.
- Audio/Visual Courses — A/V courses are designed for students who wish to learn computer fundamentals at a self-paced rate. They are portable, self-contained, and modular in format. Among the A/V courses offered are Introduction to Minicomputers, Introduction to the PDP-11, and Introduction to Data Communications Concepts.

Standard courses are given regularly at DIGITAL's training centers. However, other options are available for customers with special needs:

- On-site courses — If you have a group of individuals to train, Educational Services can conduct courses at any convenient location, such as your company offices. On-site instruction eliminates travel expenses, and allows DIGITAL instructors to emphasize points of particular value to your application and operation.
- Custom courses — If you have a unique application, Educational Services can create a custom course tailored to your needs and schedule. These courses can be modifications of existing courses or completely new programs.

VAX-11/780 Specific Courses

DIGITAL's Educational Services offers a comprehensive series of courses to provide VAX-specific training to all levels of users. These range from introductory material describing the VAX-11/780 hardware and operating systems concepts to advanced courses designed for high-level systems programmers. The diagram below illustrates a typical curriculum for three kinds of VAX-11/780 users: the system manager, a high-level language programmer, and a MACRO programmer.

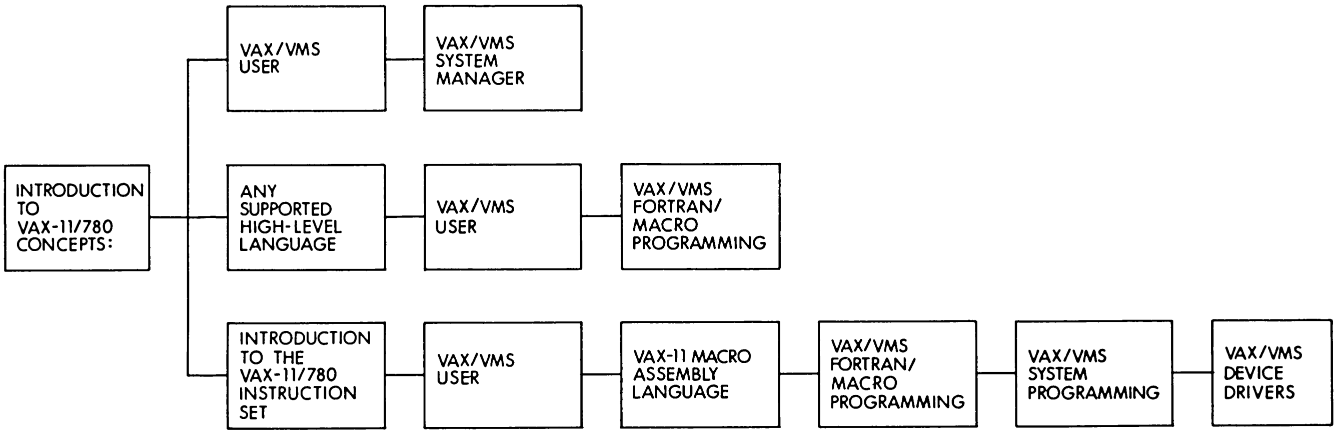

INTRODUCTION TO VAX-11/780 CONCEPTS:
VAX/VMS USER
VAX/VMS SYSTEM MANAGER
ANY SUPPORTED HIGH-LEVEL LANGUAGE
VAX/VMS USER
VAX/VMS FORTRAN/ MACRO PROGRAMMING
INTRODUCTION TO THE VAX-11/780 INSTRUCTION SET
VAX/VMS USER
VAX-11 MACRO ASSEMBLY LANGUAGE
VAX/VMS FORTRAN/ MACRO PROGRAMMING
VAX/VMS SYSTEM PROGRAMMING
VAX/VMS DEVICE DRIVERS

Figure 10-1
VAX-11/70 Courses

Following are brief descriptions of the courses shown in Figure 10-1.

Introduction to VAX-11/780 Concepts: Basic hardware and operating system components.

Introduction to VAX-11/780 Instruction Set: Introduction to addressing modes and use and coding of VAX-11/780 instructions.

VAX/VMS User: Topics useful in application program development. Editors, language translators, linkers, and utilities; VAX/VMS Command Language.

VAX/VMS System Manager: Topics useful in management of VAX/VMS system; system generation, updating, and tuning, authorization of users, error analysis and batch and I/O queue management.

VAX-11 MACRO Assembly Language: Layout conventions, assembler directives, libraries, linker, debugging techniques; advanced programming techniques.

VAX/VMS FORTRAN/MACRO Programming: Features and use of the system services provided by VAX/VMS. Discussion of run-time library, condition and exit handlers, RMS structures and directives, and interprocess commmunication. Scientific and real-time orientation.

VAX/VMS System Programming: Detailed study of VAX/VMS operating system, including paging, swapping, scheduling, I/O operations, job controller, and management of batch and spooling jobs. Techniques for optimizing application programs, improving system throughput, and adding custom software.

VAX/VMS Device Drivers: Techniques for writing and incorporating a device driver into the VMS operating system. Course is particularly geared towards non-standard peripheral devices.

Additional information concerning course content and availability of training credits may be obtained from your sales representative.

DECUS

DECUS, Digital Equipment Computer Users Society, is the largest and most active user group in the computer industry. It is supported by DIGITAL, but actively controlled by individuals who have purchased, leased, have on-order, or use a DIGITAL computer. Membership is free and voluntary.

The goals of DECUS are to:

- Advance the art of computation through mutual education and exchange of ideas and information.
- Establish standards and provide channels to facilitate the exchange of computer programs.
- Provide feedback to DIGITAL on hardware and software needs.
- Advance the effective utilization of DIGITAL computers, peripherals, and software by promoting the interchange of information concerning their use.

To further these goals, DECUS holds regular symposia, maintains a program library, publishes newsletters and symposia proceedings, and has a number of special interest and local subgroups:

- Program Library — Programs submitted by users are available to all DECUS members. Last year, the library distributed over 80,000 programs.
- Symposia — These are regularly scheduled meetings held in the United States, Canada, Europe, and Australia. DIGITAL personnel and DECUS members attend these meetings and present papers which are available to all members.
- Subgroups — Special interest groups focus on operating systems, languages, processors, and applications; in addition, numerous local users groups exist worldwide.

Your sales representative can provide additional details and the necessary application forms.

COMPUTER SPECIAL SYSTEMS

DIGITAL's Computer Special Systems (CSS) group provides solutions for special needs. You may contract this group, which includes analysts, engineers, programmers, and manufacturing specialists, to obtain hardware and software tailored to your application.

CSS builds hardware, software, and system components that supplement DIGITAL's standard product mix. Products and systems are analyzed, designed, and implemented according to your goals and requirements, and may range from simple processor interfaces to complete turnkey hardware and software systems. CSS services are available on a fixed-price contract basis.

- Special hardware — CSS interfaces DIGITAL hardware with that of other manufacturers; modifies standard hardware, and designs and builds new equipment. All special hardware can be manufactured in any quantity; spares and support are guaranteed.
- Special Software — CSS designs and produces diagnostic, systems, and applications software. They modify and expand standard DIGITAL software systems, or build new software according to your needs. When required, CSS will supply software on a turn-key basis.
- Special Systems — CSS will build complete hardware and software solutions for your special application. CSS project managers oversee the analysis, design, and implementation of the system. They work with you to ensure proper installation and start-up.
- Support — CSS solutions receive DIGITAL's standard support. Hardware is supported by DIGITAL's extensive Field Service organization, and training is available on all aspects of CSS systems.

For information on how you may be able to take advantage of the services provided by CSS, contact your sales representative.

CUSTOMER FINANCING

To simplify the financial problems in acquiring a new computer, DIGITAL provides leasing and financial counseling. The Customer Finance Department is organized to help you acquire a DIGITAL system using a lease, conditional sale, or similar financing agreement, rather than through outright cash purchase.

For private organizations or commercial businesses, DIGITAL has developed a program, known as DIGITAL Leasing, with the U.S. Leasing Corporation of San Francisco. DIGITAL Leasing is a division of U.S. Leasing committed solely to the financing of DIGITAL computers. Representatives are located in or near many of the DIGITAL District Sales Offices.

Federal, state, and local government agencies have special contractual needs, and in some cases can benefit from special tax privileges. For example, a state or municipal agency qualifies for special interest rates on Conditional Sales Agreements significantly below those charged to commercial customers; the interest income is free from federal, and in some cases state, income taxes.

The following three types of financing are available: Full Payout Lease, Conditional Sales Agreement, and Federal Government Lease to Ownership Agreement.

- Full Payout Lease — This financing is used primarily by commercial customers. It involves a non-cancelable three to seven year term, usually with a 10% purchase option at the end. No down payment is required, and title remains with the lessor. Lease payment schedules are flexible, and can be tailored to your needs.
- Conditional Sales Agreement — This type of financing is used primarily by non-profit institutions, state and local governments. It also involves a non-cancelable three to seven year term. Title passes to the customer, but DIGITAL retains a security interest. The customer owns the equipment free and clear at the end of the term. Fiscal funding provisions are available for state and local governments.
- Federal Government Lease to Ownership Agreement — This financing is available only to approved federal government agencies. It involves a three to five year term cancelable at the end of each fiscal year and at the government's convenience. Ownership passes to the customer at the end of the term.

DIGITAL's customer financing group can provide financial counseling to help decide which arrangement is best for you. For more information, contact your sales representative.

OTHER SERVICES

The following services are provided for customers who wish to perform their own maintenance, or who want to supplement their system configuration by ordering additional components.

Software Components

Software Components — software, hardware and software manuals, engineering drawings and print sets, diagnostic listings, and source listings — are available for DIGITAL software users and can be ordered from the DIGITAL Software Distribution Center in Maynard, Massachusetts. You may obtain a catalog of available materials from your salesperson.

Customer Spares

DIGITAL's Customer Spares group supports those customers who choose to perform their own computer maintenance. Customer Spares' inventory includes standard loose-piece component and module spares, tool kits, and required test equipment. In addition, Customer Spares provides comprehensive spares kits for the majority of installed processors, tape and disk drives, terminals, and other peripherals — currently 260 devices. All spares are produced directly by volume manufacturing and are inspected to the same rigid criteria as the original product.

Recommended spares lists are available for individual devices or entire systems. Maintenance documentation is also available (by subscription) and includes all standard documents (except engineering drawings) used in the maintenance of DIGITAL's hardware. Spares specialists are located at many of the DIGITAL district offices and can assist you in selecting and maintaining proper spares inventories. Contact your local DIGITAL sales office for more information and for literature that further describes Customer Spares products and services.

DIGITAL Supplies

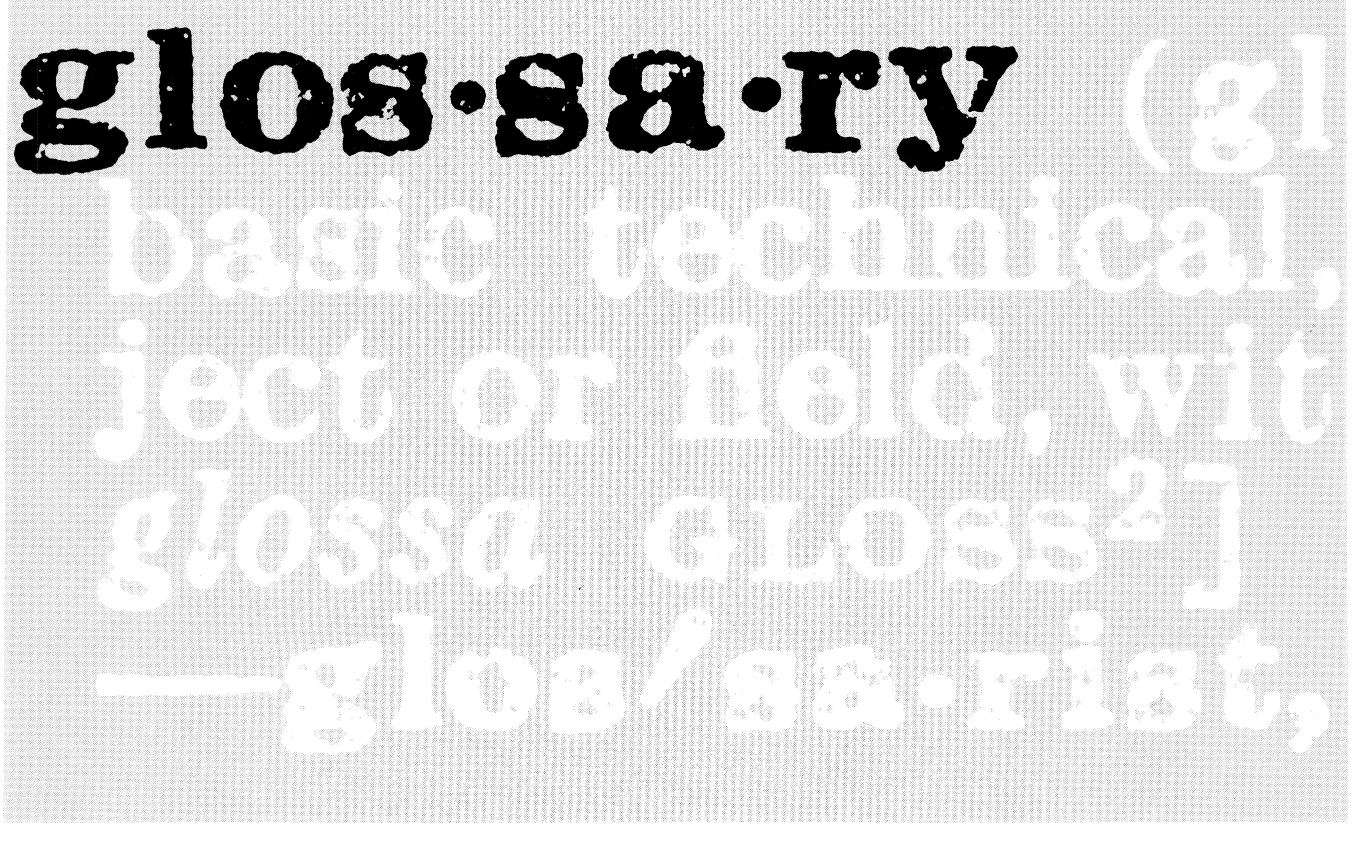
DIGITAL's Supplies Group maintains a complete line of operating computer supplies and accessories specifically designed for use with DIGITAL computer systems. You may order such supplies as magnetic disk media (disk packs, floppy disks, etc.), nylon and mylar ribbons, and acrylic filters for use on video terminals. In addition, you may order custom-configured cabinetry to store and protect supplies. Supplies specialists are located at many of the district offices and can help you plan your supply needs. Contact your local sales office for more information.

Logic Products

The Logic Products Group is your supplier for add-on equipment: power supplies, cable assemblies, line interfaces, terminals, ribbons, and paper. All logic products are described in the *Logic Handbook* and are listed in the *Direct Sales Catalog*. Ask your sales representative to help you obtain copies. You may order products directly from the catalog, or you may place your order through your local sales office.

BLANK

The Glossary



glos·sa·ry (gl
basic technical,
ject or field, wit
glossa GLOSS²]
—glos/sa·rist,

BLANK

abort An exception that occurs in the middle of an instruction and potentially leaves the registers and memory in an indeterminate state, such that the instruction can not necessarily be restarted.

absolute indexed mode An indexed addressing mode in which the base operand specifier is addressed in absolute mode.

absolute mode In absolute mode addressing, the PC is used as the register in autoincrement deferred mode. The PC contains the address of the location containing the actual operand.

absolute time Time values expressing a specific date (month, day, and year) and time of day. Absolute time values are always expressed in the system as positive numbers.

access mode 1. Any of the four processor access modes in which software executes. Processor access modes are, in order from most to least privileged and protected: kernel (mode 0), executive (mode 1), supervisor (mode 2), and user (mode 3). When the processor is in kernel mode, the executing software has complete control of, and responsibility for, the system. When the processor is in any other mode, the processor is inhibited from executing privileged instructions. The Processor Status Longword contains the current access mode field. The operating system uses access modes to define protection levels for software executing in the context of a process. For example, the executive runs in kernel and executive mode and is most protected. The command interpreter is less protected and runs in supervisor mode. The debugger runs in user mode and is not more protected than normal user programs. 2. See record access mode.

access type 1. The way in which the processor accesses instruction operands. Access types are: read, write, modify, address, and branch. 2. The way in which a procedure accesses its arguments.

access violation An attempt to reference an address that is not mapped into virtual memory or an attempt to reference an address that is not accessible by the current access mode.

account name A string that identifies a particular account used to accumulate data on a job's resource use. This name is the user's accounting charge number, not the user's UIC.

address A number used by the operating system and user software to identify a storage location. See also virtual address and physical address.

address access type The specified operand of an instruction is not directly accessed by the instruction. The address of the specified operand is the actual instruction operand. The context of the address calculation is given by the data type of the operand.

addressing mode The way in which an operand is specified; for example, the way in which the effective address of an instruction operand is calculated using the general registers. The basic general register addressing modes are called: register, register deferred, autoincrement, autoincrement deferred, autodecrement, displacement, and displacement deferred. In addition, there are six indexed addressing modes using two general registers, and literal mode addressing. The PC addressing modes are called: immediate (for register deferred mode using the PC), absolute (for autoincrement deferred mode using the PC), and branch.

address space The set of all possible addresses available to a process. Virtual address space refers to the set of all possible virtual addresses. Physical address space refers to the set of all possible physical addresses sent out on the SBI.

allocate a device To reserve a particular device unit for exclusive use. A user process can allocate a device only when that device is not allocated by any other process.

alphanumeric character An upper or lower case letter (A-Z, a-z), a dollar sign ($), an underscore (_), or a decimal digit (0-9).

American Standard Code for Information Interchange (ASCII) A set of 8-bit binary numbers representing the alphabet, punctuation, numerals, and other special symbols used in text representation and communications protocol.

Ancillary Control Process (ACP) A process that acts as an interface between user software and an I/O driver. An ACP provides functions supplemental to those performed in the driver, such as file and directory management. Three examples of ACPs are: the Files-11 ACP (F11ACP), the magnetic tape ACP (MTACP), and the networks ACP (NETACP).

Argument Pointer General register 12 (R12). By convention, AP contains the address of the base of the argument list for procedures initiated using the CALL instructions.

assign a channel To establish the necessary software linkage between a user process and a device unit before a user process can transfer any data to or from that device. A user process requests the system to assign a channel and the system returns a channel number.

asynchronous record operation A mode of record processing in which a user program can continue to execute after issuing a record retrieval or storage request without having to wait for the request to be fulfilled.

Asynchronous System Trap A software-simulated interrupt to a user-defined service routine. ASTs enable a user process to be notified asynchronously with respect to its execution of the occurrence of a specific event. If a user process has defined an AST routine for an event, the system interrupts the process and executes the AST routine when that event occurs. When the AST routine exits, the system resumes the process at the point where it was interrupted.

Asynchronous System Trap level (ASTLVL) A value kept in an internal processor register that is the highest access mode for which an AST is pending. The AST does not occur until the current access mode drops in priority (raises in numeric value) to a value greater than or equal to

ASTLVL. Thus, an AST for an access mode will not be serviced while the processor is executing in a higher priority access mode.

authorization file See user authorization file.

autodecrement indexed mode An indexed addressing mode in which the base operand specifier uses autodecrement mode addressing.

autodecrement mode In autodecrement mode addressing, the contents of the selected register are decremented, and the result is used as the address of the actual operand for the instruction. The contents of the register are decremented according to the data type context of the register: 1 for byte, 2 for word, 4 for longword and floating, 8 for quadword and double floating.

autoincrement deferred indexed mode An indexed addressing mode in which the base operand specifier uses autoincrement deferred mode addressing.

autoincrement deferred mode In autoincrement deferred mode addressing, the specified register contains the address of a longword which contains the address of the actual operand. The contents of the register are incremented by 4 (the number of bytes in a longword). If the PC is used as the register, this mode is called absolute mode.

autoincrement indexed mode An indexed addressing mode in which the base operand specifier uses autoincrement mode addressing.

autoincrement mode In autoincrement mode addressing, the contents of the specified register are used as the address of the operand, then the contents of the register are incremented by the size of the operand.

balance set The set of all process working sets currently resident in physical memory. The processes whose working sets are in the balance set have memory requirements that balance with available memory. The balance set is maintained by the system swapper process.

base operand address The address of the base of a table or array referenced by index mode addressing.

base operand specifier The register used to calculate the base operand address of a table or array referenced by index mode addressing.

base priority The process priority that the system assigns a process when it is created. The scheduler never schedules a process below its base priority. The base priority can be modified only by the system manager or the process itself.

base register A general register used to contain the address of the first entry in a list, table, array, or other data structure.

binding See linking.

bit string See variable-length bit field.

block 1. The smallest addressable unit of data that the specified device can transfer in an I/O operation (512 contiguous bytes for most disk devices). 2. An arbitrary number of contiguous bytes used to store logically related status, control, or other processing information.

block I/O A data accessing technique in which the program manipulates the blocks (physical records) that make up a file, instead of its logical records.

bootstrap block A block in the index file on a system disk that contains a program that can load the operating system into memory and start its execution.

branch access type An instruction attribute which indicates that the processor does not reference an operand address, but that the operand is a branch displacement. The size of the branch displacement is given by the data type of the operand.

branch mode In branch addressing mode, the instruction operand specifier is a signed byte or word displacement. The displacement is added to the contents of the updated PC (which is the address of the first byte beyond the displacement), and the result is the branch address.

bucket A storage structure of 1 to 32 blocks that is used to store and transfer blocks of data in files with a relative file organization.

bucket locking A facility that prevents access to any record in a bucket by more than one user until that user releases the bucket.

buffered I/O See system buffered I/O.

bug check The operating system's internal diagnostic check. The system logs the failure and crashes the system.

byte A byte is eight contiguous bits starting on an addressable byte boundary. Bits are numbered from the right, 0 through 7, with bit 0 the low-order bit. When interpreted arithmetically, a byte is a two's complement integer with significance increasing from bits 0 through 6. Bit 7 is the sign bit. The value of the signed integer is in the range -128 to 127 decimal. When interpreted as an unsigned integer, significance increases from bits 0 through 7 and the value of the unsigned integer is in the range 0 to 255 decimal. A byte can be used to store one ASCII character.

cache memory A small, high-speed memory placed between slower main memory and the processor. A cache increases effective memory transfer rates and processor speed. It contains copies of data recently used by the processor, and fetches several bytes of data from memory in anticipation that the processor will access the next sequential series of bytes.

call frame See stack frame.

call instructions The processor instructions CALLG (Call Procedure with General Argument List) and CALLS (Call Procedure with Stack Argument List).

call stack The stack, and conventional stack structure, used during a procedure call. Each access mode of each process context has one call stack, and interrupt service context has one call stack.

channel A logical path connecting a user process to a physical device unit. A user process requests the operating system to assign a channel to a device so the process can transfer data to or from that device.

character A symbol represented by an ASCII code. See also alphanumeric character.

character string A contiguous set of bytes. A character string is identified by two attributes: an address and a length. Its address is the address of the byte containing the first character of the string. Subsequent characters are stored in bytes of increasing addresses. The length is the number of characters in the string.

character string descriptor A quadword data structure used for passing character data (strings). The first word of the quadword contains the length of the character string. The second word can contain type information. The remaining longword contains the address of the string.

cluster 1. A set of contiguous blocks that is the basic unit of space allocation on a Files-11 disk volume. 2. A set of pages brought into memory in one paging operation. 3. An event flag cluster.

command An instruction, generally an English word, typed by the user at a terminal or included in a command file which requests the software monitoring a terminal or reading a command file to perform some well-defined activity. For example, typing the COPY command requests the system to copy the contents of one file into another file.

command file A file containing command strings. See also command procedure.

command interpreter Procedure-based system code that executes in supervisor mode in the context of a process to receive, syntax check, and parse commands typed by the user at a terminal or submitted in a command file.

command parameter The positional operand of a command delimited by spaces, such as a file specification, option, or constant.

command procedure A file containing commands and data that the command interpreter can accept in lieu of the user typing the commands individually on a terminal.

command string A line (or set of continued lines), normally terminated by typing the carriage return key, containing a command and, optionally, information modifying the command. A complete command string consists of a command, its qualifiers, if any, and its parameters (file specifications, for example), if any, and their qualifiers, if any.

common A FORTRAN term for a program section that contains only data.

common event flag cluster A set of 32 event flags that enables cooperating processes to post event notification to each other. Common event flag clusters are created as they are needed. A process can associate with up to two common event flag clusters.

compatibility mode A mode of execution that enables the central processor to execute non-privileged PDP-11 instructions. The operating system supports compatibility mode execution by providing an RSX-11M programming environment for an RSX-11M task image. The operating system compatibility mode procedures reside in the control region of the process executing a compatibility mode image. The procedures intercept calls to the RSX-11M executive and convert them to the appropriate operating system functions.

condition An exception condition detected and declared by software. For example, see failure exception mode.

condition codes Four bits in the Processor Status Word that indicate the results of previously executed instructions.

condition handler A procedure that a process wants the system to execute when an exception condition occurs. When an exception condition occurs, the operating system searches for a condition handler and, if found, initiates the handler immediately. The condition handler may perform some action to change the situation that caused the exception condition and continue execution for the process that incurred the exception condition. Condition handlers execute in the context of the process at the access mode of the code that incurred the exception condition.

condition value A 32-bit quantity that uniquely identifies an exception condition.

context The environment of an activity. See also process context, hardware context, and software context.

context indexing The ability to index through a data structure automatically because the size of the data type is known and used to determine the offset factor.

context switching Interrupting the activity in progress and switching to another activity. Context switching occurs as one process after another is scheduled for execution. The operating system saves the interrupted process' hardware context in its hardware process control block (PCB) using the Save Process Context instruction, loads another process' hardware PCB into the hardware context using the Load Process Context instruction, scheduling that process for execution.

continuation character A hyphen at the end of a command line signifying that the command string continues on to the next command line.

console The manual control unit integrated into the central processor. The console includes an LSI-11 microprocessor and a serial line interface connected to a hard copy terminal. It enables the operator to start and stop the system, monitor system operation, and run diagnostics.

console terminal The hard copy terminal connected to the central processor console.

control region The highest-addressed half of per-process space (the P1 region). Control region virtual addresses refer to the process-related information used by the system to control the process, such as: the kernel, executive, and supervisor stacks, the permanent I/O channels, exception vectors, and dynamically used system procedures (such as the command interpreter and RSX-11M programming environment compatibility mode procedures). The user stack is also normally found in the control region, although it can be relocated elsewhere.

Control Region Base Register (P1BR) The processor register, or its equivalent in a hardware process control block, that contains the base virtual address of a process control region page table.

Control Region Length Register (P1LR) The processor register, or its equivalent in a hardware process control block, that contains the number of non-existent page table entries for virtual pages in a process control region.

copy-on-reference A method used in memory management for sharing data until a process accesses it, in which case it is copied before the access. Copy-on-reference allows sharing of the initial values of a global section whose pages have read/write access but contain pre-initialized data available to many processes.

counted string A data structure consisting of a byte-sized length followed by the string. Although a counted string is not used as a procedure argument, it is a convenient representation in memory.

current access mode The processor access mode of the currently executing software. The Current Mode field of the Processor Status Longword indicates the access mode of the currently executing software.

cylinder The tracks at the same radius on all recording surfaces of a disk.

data base 1. All the occurrences of data described by a data base management system. 2. A collection of related data structures.

data structure Any table, list, array, queue, or tree whose format and access conventions are well-defined for reference by one or more images.

data type In general, the way in which bits are grouped and interpreted. In reference to the processor instructions, the data type of an operand identifies the size of the operand and the significance of the bits in the operand. Operand data types include: byte, word, longword, and quadword integer, floating and double floating, character string, packed decimal string, and variable-length bit field.

deferred echo Refers to the fact that terminal echoing does not occur until a process is ready to accept input entered by type ahead.

delta time A time value expressing an offset from the current date and time. Delta times are always expressed in the system as negative numbers whose absolute value is used as an offset from the current time.

demand zero page A page, typically of an image stack or buffer area, that is initialized to contain all zeros when dynamically created in memory as a result of a page fault. This feature eliminates the waste of disk space that would otherwise be required to store blocks (pages) that contain only zeros.

descriptor A data structure used in calling sequences for passing argument types, addresses and other optional information. See character string descriptor.

detached process A process that has no owner. The parent process of a tree of subprocesses. Detached processes are created by the job controller when a user logs on the system or a when a batch job is initiated. The job controller does not own the user processes it creates; these processes are therefore detached.

device The general name for any physical terminus or link connected to the processor that is capable of receiving, storing, or transmitting data. Card readers, line printers, and terminals are examples of record-oriented devices. Magnetic tape devices and disk devices are examples of mass storage devices. Terminal line interfaces and interprocessor links are examples of communications devices.

device interrupt An interrupt received on interrupt priority level 16 through 23. Device interrupts can be requested only by devices, controllers, and memories.

device name The field in a file specification that identifies the device unit on which a file is stored. Device names also include the mnemonics that identify an I/O peripheral device in a data transfer request. A device name consists of a mnemonic followed by a controller identification letter (if applicable), followed by a unit number (if applicable). A colon (:) separates it from following fields.

device queue See spool queue.

device register A location in device controller logic used to request device functions (such as I/O transfers) and/or report status.

device unit One drive, and its controlling logic, of a mass storage device system. A mass storage system can have several drives connected to it.

diagnostic A program that tests logic and reports any faults it detects.

direct I/O An I/O operation in which the system locks the pages containing the associated buffer in memory for the duration of the I/O operation. The I/O transfer takes place directly from the process buffer. Contrast with system buffered I/O.

direct mapping cache A cache organization in which only one address comparision is needed to locate any data in the cache because any block of main memory data can be placed in only one possible position in the cache. Contrast with fully associative cache.

directory A file used to locate files on a volume that contains a list of file names (including type and version number) and their unique internal identifications.

directory name The field in a file specification that identifies the directory file in which a file is listed. The directory name begins with a left bracket ([or <) and ends with a right bracket (] or >).

displacement deferred indexed mode An indexed addressing mode in which the base operand specifier uses displacement deferred mode addressing.

displacement deferred mode In displacement deferred mode addressing, the specifier extension is a byte, word, or longword displacement. The displacement is sign extended to 32 bits and added to a base address obtained from the specified register. The result is the address of a longword which contains the address of the actual operand. If the PC is used as the register, the updated contents of the PC are used as the base address. The base address is the address of the first byte beyond the specifier extension.

displacement indexed mode An indexed addressing mode in which the base operand specifier uses displacement mode addressing.

displacement mode In displacement mode addressing, the specifier extension is a byte, word, or longword displacement. The displacement is sign extended to 32 bits and added to a base address obtained from the specified register. The result is the address of the actual operand. If the PC is used as the register, the updated contents of the PC are used as the base address. The base address is the address of the first byte beyond the specifier extension.

double floating datum Eight contiguous bytes (64 bits), starting on an addressable byte boundary, which are interpreted as containing a floating point number. The bits are labeled from right to left, 0 to 63. A four-word floating point number is identified by the address of the byte containing bit 0. Bit 15 contains the sign of the number. Bits 14 through 7 contain the excess 128 binary exponent. Bits 63 through 16 and 6 through 0 contain a normalized 56-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of decreasing significance go from 6 through 0, 31 through 16, 47 through 32, then 63 through 48. Exponent values of 1 through 255 in the 8-bit exponent field represent true binary exponents of -128 to 127. An exponent value of 0 together with a sign bit of 0 represent a floating value of 0. An exponent value of 0 with a sign bit of 1 is a reserved representation; floating point instructions processing this value return a reserved operand fault. The value of a floating datum is in the approximate range (+ or -) 0.29×10^{-38} to 1.7×10^{38}. The precision is approximately one part in 2^{55} or sixteen decimal digits.

drive The electro-mechanical unit of a mass storage device system on which a recording medium (disk cartridge, disk pack, or magnetic tape reel) is mounted.

driver The set of code that handles physical I/O to a device.

dynamic access A technique in which a program switches from one record access mode to another while processing a file.

echo A terminal handling characteristic in which the characters typed by the terminal user on the keyboard are also displayed on the screen or printer.

effective address The address obtained after indirect or indexing modifications are calculated.

entry mask A word whose bits represent the registers to be saved or restored on a subroutine or procedure call using the call and return instructions.

entry point A location that can be specified as the object of a call. It contains an entry mask and exception enables known as the entry point mask.

equivalence name The string associated with a logical name in a logical name table. An equivalence name can be for example, a device name, another logical name, or a logical name concatenated with a portion of a file specification.

error logger A system process that empties the error log buffers and writes the error messages into the error file. Errors logged by the system include memory system errors, device errors and timeouts, and interrupts with invalid vector addresses.

escape sequence An escape is a transition from the normal mode of operation to a mode outside the normal mode. An escape character is the code that indicates the transition from normal to escape mode. An escape sequence refers to the set of character combinations starting with an escape character that the terminal transmits without interpretation to the software set up to handle escape sequences.

event A change in process status or an indication of the occurrence of some activity that concerns an individual process or cooperating processes. An incident reported to the scheduler that affects a process' ability to execute. Events can be synchronous with the process' execution (a wait request), or they can be asynchronous (I/O completion). Some other events include: swapping; wake request; page fault.

event flag A bit in an event flag cluster that can be set or cleared to indicate the occurrence of the event associated with that flag. Event flags are used to synchronize activities in a process or among many processes.

event flag cluster A set of 32 event flags which are used for event posting. Four clusters are defined for each process: two process-local clusters, and two common event flag clusters. Of the process-local flags, eight are reserved for system use.

exception An event detected by the hardware (other than an interrupt or jump, branch, case, or call instruction) that changes the normal flow of instruction execution. An exception is always caused by the execution of an instruction or set of instructions (whereas an interrupt is caused by an activity in the system independent of the current instruction). There are three types of hardware exceptions: traps, faults, and aborts. Examples are: attempts to execute a privileged or reserved instruction, trace traps, compatibility mode faults, breakpoint instruction execution, and arithmetic traps such as overflow, underflow, and divide by zero.

exception condition A hardware- or software-detected event other than an interrupt or jump, branch, case, or call instruction that changes the normal flow of instruction execution.

exception dispatcher An operating system procedure that searches for a condition handler when an exception condition occurs. If no exception handler is found for the exception or condition, the image that incurred the exception is terminated.

exception enables See trap enables.

exception vector See vector.

executable image An image that is capable of being run in a process. When run, an executable image is read from a file for execution in a process.

executive The generic name for the collection of procedures included in the operating system software that provides the basic control and monitoring functions of the operating system.

executive mode The second most privileged processor access mode (mode 1). The record management services (RMS) and many of the operating system's programmed service procedures execute in executive mode.

exit An image exit is a rundown activity that occurs when image execution terminates either normally or abnormally. Image rundown activities include deassigning I/O channels and disassociation of common event flag clusters. Any user- or system-specified exit handlers are called.

exit handler A procedure executed when an image exits. An exit handler enables a procedure that is not on the call stack to gain control and clean up procedure-own data bases before the actual image exit occurs.

extended attribute block (XAB) An RMS user data structure that contains additional file attributes beyond those expressed in the file access block (FAB), such as boundary types (aligned on cylinder, logical block number, virtual block number) and file protection information.

extension The amount of space to allocate at the end of a file each time a sequential write exceeds the allocated length of the file.

extent The contiguous area on a disk containing a file or a portion of a file. Consists of one or more clusters.

failure exception mode A mode of execution selected by a process indicating that it wants an exception condition declared if an error occurs as the result of a system service call. The normal mode is for the system service to return an error status code for which the process must test.

fault A hardware exception condition that occurs in the middle of an instruction and that leaves the registers and memory in a consistent state, such that elimination of the fault and restarting the instruction will give correct results.

field 1. See variable-length bit field. 2. A set of contiguous bytes in a logical record.

file access block (FAB) An RMS user data structure that represents a request for data operations related to the file as a whole, such as OPEN, CLOSE, or CREATE.

file header A block in the index file describing a file on a Files-11 disk structure. The file header identifies the locations of the file's extents. There is a file header for every file on the disk.

file name The field preceding a file type in a file specification that contains a 1- to 9-character logical name for a file.

filename extension See file type.

file organization The particular file structure used to record the data comprising a file on a mass storage medium. RMS file organizations are: sequential, relative, direct, and indexed.

Files-11 The name of the on-disk structure used by the RSX-11, IAS and VAX operating systems. Volumes created under this structure are transportable between these operating systems.

file specification A unique name for a file on a mass storage medium. It identifies the node, the device, the directory name, the file name, the file type, and the version number under which a file is stored.

file structure The way in which the blocks forming a file are distributed on a disk or magnetic tape to provide a physical accessing technique suitable for the way in which the data in the file is processed.

file system A method of recording, cataloging, and accessing files on a volume.

file type The field in a file specification that is preceded by a period or dot (.) and consists of a zero- to three-character type identification. By convention, the type identifies a generic class of files that have the same use or characteristics, such as ASCII text files, binary object files, etc.

fixed control area An area associated with a variable length record available for controlling or assisting record access operations. Typical uses include line numbers and printer format control information.

fixed length record format A file format in which all records have the same length.

floating (point) datum Four contiguous bytes (32 bits) starting on an addressable byte boundary. The bits are labeled from right to left from 0 to 31. A two-word floating point number is identified by the address of the byte containing bit 0. Bit 15 contains the sign of the number. Bits 14 through 7 contain the excess 128 binary exponent. Bits 31 through 16 and 6 through 0 contain a normalized 24-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of decreasing significance go from bit 6 through 0, then 31 through 16. Exponent values of 1 through 255 in the 8-bit exponent field represent true binary exponents of -128 to 127. An exponent value of 0 together with a sign bit of 0 represent a floating value of 0. An exponent value of 0 with a sign bit of 1 is a reserved representation; floating point instructions processing this value return a reserved operand fault. The value of a floating datum is in the approximate range (+ or -) 0.29×10^{-38} to 1.7×10^{38}. The precision is approximately one part in 2^{23} or seven decimal digits.

foreign volume Any volume other than a Files-11 formatted volume which may or may not be file structured.

fork process A dynamically created system process such as a process that executes device driver code or the timer process. Fork processes have minimal context. Fork processes are scheduled by the hardware rather than by the software. The timer process is dispatched directly by software interrupt. I/O driver processes are dispatched by a fork dispatcher. Fork processes execute at software interrupt levels and are dispatched for execution immediately. Fork processes remain resident until they terminate.

frame pointer General register 13 (R13). By convention, FP contains the base address of the most recent call frame on the stack.

fully associative cache A cache organization in which any block of data from main memory can be placed anywhere in the cache. Address comparision must take place against each block in the cache to find any particular block. Constrast with direct mapping cache.

general register Any of the sixteen 32-bit registers used as the primary operands of the native mode instructions. The general registers include 12 general purpose registers which can be used as accumulators, as counters, and as pointers to locations in main memory, and the Frame Pointer (FP), Argument Pointer (AP), Stack Pointer (SP), and Program Counter (PC) registers.

generic device name A device name that identifies the type of device but not a particular unit; a device name in which the specific controller and/or unit number is omitted.

giga Metric term used to represent the number 1 followed by nine zeros.

global page table The page table containing the master page table entries for global sections.

global section A data structure (e.g., FORTRAN global common) or sharable image section potentially available to all processes in the system. Access is protected by privilege and/or group number of the UIC.

global symbol A symbol defined in a module that is potentially available for reference by another module. The linker resolves (matches references with definitions) global symbols. Contrast with local symbol.

global symbol table (GST) In a library, an index of strongly defined global symbols used to access the modules defining the global symbols. The linker will also put global symbol tables into an image. For example, the linker appends a global symbol table to executable images that are intended to run under the symbolic debugger, and it appends a global symbol table to all sharable images.

group 1. A set of users who have special access privileges to each other's directories and files within those directories (unless protected otherwise), as in the context "system, owner, group, world," where group refers to all members of a particular owner's group. 2. A set of jobs (processes and their subprocesses) who have access privileges to a group's common event flags and logical name tables, and may have mutual process controlling privileges, such as scheduling, hibernation, etc.

group number The first number in a User Identification Code (UIC).

hardware context The values contained in the following registers while a process is executing: the Program Counter (PC); the Processor Status Longword (PSL); the 14 general registers (R0 through R13); the four processor registers (P0BR, P0LR, P1BR and P1LR) that describe the process virtual address space; the Stack Pointer (SP) for the current access mode in which the processor is executing; plus the contents to be loaded in the Stack Pointer for every access mode other than the current access mode. While a process is executing, its hardware context is continually being updated by the processor. While a process is not executing its hardware context is stored in its hardware PCB.

hardware process control block (PCB) A data structure known to the processor that contains the hardware context when a process is not executing. A process' hardware PCB resides in its process header.

hibernation A state in which a process is inactive, but known to the system with all of its current status. A hibernating process becomes active again when a wake request is issued. It can schedule a wake request before hibernating, or another process can issue its wake request. A hibernating process also becomes active for the time sufficient to service any AST it may receive while it is hibernating. Contrast with suspension.

home block A block in the index file that contains the volume identification, such as volume label and protection.

image An image consists of procedures and data that have been bound together by the linker. There are three types of images: executable, sharable, and system.

image activator A set of system procedures that prepare an image for execution. The image activator establishes the memory management data structures required both to map the image's virtual pages to physical pages and to perform paging.

image exit See exit.

image I/O segment That portion of the control region that contains the RMS internal file access blocks (IFAB) and I/O buffers for the image currently being executed by a process.

image name The file name of the file in which an image is stored.

image privileges The privileges assigned to an image when it is linked. See process privileges.

image section (isect) A group of program sections (psects) with the same attributes (such as read-only access, read/write access, absolute, relocatable, etc.) that is the unit of virtual memory allocation for an image.

immediate mode In immediate mode addressing, the PC is used as the register in autoincrement mode addressing.

indexed addressing mode In indexed mode addressing, two registers are used to determine the actual instruction operand: an index register and a base operand specifier. The contents of the index register are used as an index (offset) into a table or array. The base operand specifier supplies the base address of the array (the base operand address or BOA). The address of the actual operand is calculated by multiplying the contents of the index register by the size (in bytes) of the actual operand and adding the result to the base operand address. The addressing modes resulting from index mode addressing are formed by adding the suffix "indexed" to the addressing mode of the base operand specifier: register deferred indexed, autoincrement indexed, autoincrement deferred indexed (or absolute indexed), autodecrement indexed, displacement indexed, and displacement deferred indexed.

indexed file organization A file organization in which a file contains records and a primary key index (and optionally one or more alternate key indices) used to process the records sequentially by index or randomly by index.

index file The file on a Files-11 volume that contains the access information for all files on the volume and enables the operating system to identify and access the volume.

index file bit map A table in the index file of a Files-11 volume that indicates which file headers are in use.

index register A register used to contain an address offset.

indirect command file See command procedure.

input stream The source of commands and data. One of: the user's terminal, the batch stream, or an indirect command file.

instruction buffer An 8-byte buffer in the processor used to contain bytes of the instruction currently being decoded and to pre-fetch instructions in the instruction stream. The control logic continously fetches data from memory to keep the 8-byte buffer full.

interleaving Assigning consecutive physical memory addresses alternately between two memory controllers.

interprocess communication facility A common event flag, mailbox, or global section used to pass information between two or more processes.

interrecord gap A blank space deliberately placed between data records on the recording surface of a magnetic tape.

interrupt An event other than an exception or branch, jump, case, or call instruction that changes the normal flow of instruction execution. Interrupts are generally external to the process executing when the interrupt occurs. See also device interrupt, software interrupt, and urgent interrupt.

interrupt priority level (IPL) The interrupt level at which the processor executes when an interrupt is generated. There are 31 possible interrupt priority levels. IPL 1 is lowest, 31 highest. The levels arbitrate contention for processor service. For example, a device cannot interrupt the processor if the processor is currently executing at an interrupt priority level greater than the interrupt priority level of the device's interrupt service routine.

interrupt service routine The routine executed when a device interrupt occurs.

interrupt stack The system-wide stack used when executing in interrupt service context. At any time, the processor is either in a process context executing in user, supervisor, executive or kernel mode, or in system-wide interrupt service context operating with kernel privileges, as indicated by the interrupt stack and current mode bits in the PSL. The interrupt stack is not context switched.

interrupt stack pointer The stack pointer for the interrupt stack. Unlike the stack pointers for process context stacks, which are stored in the hardware PCB, the interrupt stack pointer is stored in an internal register.

interrupt vector See vector.

I/O driver See driver.

I/O function An I/O operation that is interpreted by the operating system and typically results in one or more physical I/O operations.

I/O function code A 6-bit value specified in a Queue I/O Request system service that describes the particular I/O operation to be performed (e.g., read, write, rewind).

I/O function modifier A 10-bit value specified in a Queue I/O Request system service that modifies an I/O function code (e.g., read terminal input no echo).

I/O lockdown The state of a page such that it can not be paged or swapped out of memory until any I/O in progress to that page is completed.

I/O rundown An operating system function in which the system cleans up any I/O in progress when an image exits.

I/O space The region of physical address space that contains the configuration registers, and device control/status and data registers.

I/O status block A data structure associated with the Queue I/O Request system service. This service optionally returns a status code, number of bytes transferred, and device- and function-dependent information in an I/O status block. It is not returned from the service call, but filled in when the I/O request completes:

job 1. A job is the accounting unit equivalent to a process and the collection of all the subprocesses, if any, that it and its subprocesses create. Jobs are classified as batch and interactive. For example, the job controller creates an interactive job to handle a user's requests when the user logs onto the system and it creates a batch job when the symbiont manager passes a command input file to it. 2. A print job.

job controller The system process that establishes a job's process context, starts a process running the LOGIN image for the job, maintains the accounting record for the job, manages symbionts, and terminates a process and its subprocesses.

job queue A list of files that a process has supplied for processing by a specific device, for example, a line printer.

kernel mode The most privileged processor access mode (mode 0). The operating system's most privileged services, such as I/O drivers and the pager, run in kernel mode.

lexical function A command language construct that the command interpreter evaluates and substitutes before it performs expression analysis on a command string. Lexical functions return information about the current process, such as UIC or default directory; and about character strings, such as length or substring locations.

librarian A program that allows the user to create, update, modify, list, and maintain object library, image library, and assembler macro library files.

library file A direct access file containing one or·more modules of the same module type.

limit The size or number of given items requiring system resources (such as mailboxes, locked pages, I/O requests, open files, etc.) that a job is allowed to have at any one time during execution, as specified by the system manager in the user authorization file. See also quota.

line number A number used to identify a line of text in a file processed by a text editor.

linker A program that reads one or more object files created by language processors and produces an executable image file, a sharable image file, or a system image file.

linking The resolution of external references between object modules used to create an image, the acquisition of referenced library routines, service entry points, and data for the image, and the assignment of virtual addresses to components of an image.

literal mode In literal mode addressing, the instruction operand is a constant whose value is expressed in a 6-bit field of the instruction. If the operand data type is byte, word, longword, or quadword, the operand is zero extended and can express values in the range 0 through 63 (decimal). If the operand data type is floating or double floating, the 6-bit field is composed of two 3-bit fields, one for the exponent and the other for the fraction. The operand is extended to floating or double floating format.

locality See program locality.

local symbol A symbol meaningful only to the module that defines it. Symbols not identified to a language processor as global symbols are considered to be local symbols. A language processor resolves (matches references with definitions) local symbols. They are not known to the linker and can not be made available to another object module. They can, however, be passed through the linker to the symbolic debugger. Contrast with global symbol.

locate mode A record access technique in which a program reads records in an RMS block buffer working storage area to reduce overhead. See also move mode.

locking a page in memory Making a page in an image ineligible for either paging or swapping. A page stays locked in memory until it is specifically unlocked.

locking a page in the working set Making a page in an image ineligible for paging out of the working set for the image. The page can be swapped when the process is swapped. A page stays locked in a working set until it is specifically unlocked.

logical block A block on a mass storage device identified using a volume-relative address rather than its physical (device-oriented) address or its virtual (file-relative) address. The blocks that constitute the volume are labeled sequentially starting with logical block 0.

logical I/O function A set of I/O operations (e.g., read and write logical block) that allow restricted direct access to device level I/O operations using logical block addresses.

logical name A user-specified name for any portion or all of a file specification. For example, the logical name INPUT can be assigned to a terminal device from which a program reads data entered by a user. Logical name assignments are maintained in logical name tables for each process, each group, and the system. A logical name can be created and assigned a value permanently or dynamically.

logical name table A table that contains a set of logical names and their equivalence names for a particular process, a particular group, or the system.

logical I/O functions A set of I/O functions that allow restricted direct access to device level I/O operations.

logical record A group of related fields treated as a unit.

longword Four contiguous bytes (32 bits) starting on an addressable byte boundary. Bits are numbered from right to left with 0 through 31. The address of the longword is the address of the byte containing bit 0. When interpreted arithmetically, a longword is a 2's complement integer with significance increasing from bit 0 to bit 30. When interpreted as a signed integer, bit 31 is the sign bit. The value of the signed integer is in the range -2,147,483,648 to 2,147,483,647. When interpreted as an unsigned integer, significance increases from bit 0 to bit 31. The value of the unsigned integer is in the range 0 through 4,294,967,295.

macro A statement that requests a language processor to generate a predefined set of instructions.

mailbox A software data structure that is treated as a record-oriented device for general interprocess communication. Communication using a mailbox is similar to other forms of device-independent I/O. Senders perform a write to a mailbox, the receiver performs a read from that mailbox. Some system-wide mailboxes are defined: the error logger and OPCOM read from system-wide mailboxes.

main memory See physical memory.

mapping window A subset of the retrieval information for a file that is used to translate virtual block numbers to logical block numbers.

mass storage device A device capable of reading and writing data on mass storage media such as a disk pack or a magnetic tape reel.

member number The second number in a user identification code that uniquely identifies that code.

memory management The system functions that include the hardware's page mapping and protection and the operating system's image activator and pager.

Memory Mapping Enable (MME) A bit in a processor register that governs address translation.

modify access type The specified operand of an instruction or procedure is read, and is potentially modified and written, during that instruction's or procedure's execution.

module 1. A portion of a program or program library, as in a *source module*, *object module*, or *image module*. 2. A board, usually made of plastic covered with an electrical conductor, on which logic devices (such as transistors, resistors, and memory chips) are mounted, and circuits connecting these devices are etched, as in a *logic module*.

Monitor Console Routine (MCR) The command interpreter in an RSX-11 system.

mount a volume 1. To logically associate a volume with the physical unit on which it is loaded (an activity accomplished by system software at the request of an operator). 2. To load or place a magnetic tape or disk pack on a drive and place the drive on-line (an activity accomplished by a system operator).

move mode A record I/O access technique in which a program accesses records in its own working storage area. See also locate mode.

mutex A semaphore that is used to control exclusive access to a region of code that can share a data structure or other resource. The mutex (mutual exclusion) semaphore ensures that only one process at a time has access to the region of code.

name block (NAM) An RMS user data structure that contains supplementary information used in parsing file specifications.

native image An image whose instructions are executed in native mode.

native mode The processor's primary execution mode in which the programmed instructions are interpreted as byte-aligned, variable-length instructions that operate on byte, word, longword, and quadword integer, floating and double floating, character string, packed decimal, and variable-length bit field data. The instruction execution mode other than compatibility mode.

network A collection of interconnected individual computer systems.

nibble The low-order or high-order four bits of a byte.

node An individual computer system in a network.

null process A small system process that is the lowest priority process in the system and takes one entire priority class. One function of the null process is to accumulate idle processor time.

numeric string A contiguous sequence of bytes representing up to 31 decimal digits (one per byte) and possibly a sign. The numeric string is specified by its lowest addressed location, its length, and its sign representation.

object module The binary output of a language processor such as the assembler or a compiler, which is used as input to the linker.

object time system (OTS) See Run Time Procedure Library.

offset A fixed displacement from the beginning of a data structure. System offsets for items within a data structure normally have an associated symbolic name used instead of the numeric displacement. Where symbols are defined, programmers always reference the symbolic names for items in a data structure instead of using the numeric displacement.

on-disk structure-1 (ODS-1) Refer to Files-11 structure Level 1.

on-disk structure-2 (ODS-2) Refer to Files-11 structure Level 2.

opcode The pattern of bits within an instruction that specify the operation to be performed.

operand specifier The pattern of bits in an instruction that indicate the addressing mode, a register and/or displacement, which, taken together, identify an instruction operand.

operand specifier type The access type and data type of an instruction's operand(s). For example, the test instructions are of read access type, since they only read the value of the operand. The operand can be of byte, word, or longword data type, depending on whether the opcode is for the TSTB (test byte), TSTW (test word), or TSTL (test longword) instruction.

Operator Communication Manager (OPCOM) A system process that is always active. OPCOM receives input from a process that wants to inform an operator of a particular status or condition, passes a message to the operator, and tracks the message.

operator's console Any terminal identified as a terminal attended by a system operator.

owner In the context "system, owner, group, world," an owner is the particular member (of a group) to which a file, global section, mailbox, or event flag cluster belongs.

owner process The process (with the exception of the job controller) or subprocess that created a subprocess.

packed decimal A method of representing a decimal number by storing a pair of decimal digits in one byte, taking advantage of the fact that only four bits are required to represent the numbers zero through nine.

packed decimal string A contiguous sequence of up to 16 bytes interpreted as a string of nibbles. Each nibble represents a digit except the low-order nibble of the highest addressed byte, which represents the sign. The packed decimal string is specified by its lowest addressed location and the number of digits.

page 1. A set of 512 contiguous byte locations used as the unit of memory mapping and protection. 2. The data between the beginning of file and a page marker, between two markers, or between a marker and the end of a file.

page fault An exception generated by a reference to a page which is not mapped into a working set.

page fault cluster size The number of pages read in on a page fault.

page frame number (PFN) The address of the first byte of a page in physical memory. The high-order 21 bits of the physical address of the base of a page.

page marker A character or characters (generally a form feed) that separates pages in a file that is processed by a text editor.

pager A set of kernel mode procedures that executes as the result of a page fault. The pager makes the page for which the fault occurred available in physical memory so that the image can continue execution. The pager and the image activator provide the operating system's memory management functions.

page table entry (PTE) The data structure that identifies the location and status of a page of virtual address space. When a virtual page is in memory, the PTE contains the page frame number needed to map the virtual page to a physical page. When it is not in memory, the page table entry contains the information needed to locate the page on secondary storage (disk).

paging The action of bringing pages of an executing process into physical memory when referenced. When a process executes, all of its pages are said to reside in virtual memory. Only the actively used pages, however, need to reside in physical memory. The remaining pages can reside on disk until they are needed in physical memory. In this system, a process is paged only when it references more pages than it is allowed to have in its working set. When the process refers to a page not in its working set, a page fault occurs. This causes the operating system's pager to read in the referenced page if it is on disk (and, optionally, other related pages depending on a cluster factor), replacing the least recently faulted pages as needed. A process pages only against itself.

parameter See command parameter.

per-process address space See process address space.

physical address The address used by hardware to identify a location in physical memory or on directly addressable secondary storage devices such as a disk. A physical memory address consists of a page frame number and the number of a byte within the page. A physical disk block address consists of a cylinder or track and sector number.

physical address space The set of all possible 30-bit physical addresses that can be used to refer to locations in memory (memory space) or device registers (I/O space).

physical block A block on a mass storage device referred to by its physical (device-oriented) address rather than a logical (volume-relative) or virtual (file-relative) address.

physical I/O functions A set of I/O functions that allow access to all device level I/O operations except maintenance mode.

physical memory The memory modules connected to the SBI that are used to store: 1) instructions that the processor can directly fetch and execute, and 2) any other data that a processor is instructed to manipulate. Also called main memory.

position dependent code Code that can execute properly only in the locations in virtual address space that are assigned to it by the linker.

position independent code Code that can execute properly without modification wherever it is located in virtual address space, even if its location is changed after it has been linked. Generally, this code uses addresssing modes that form an effective address relative to the PC.

primary vector A location that contains the starting address of a condition handler to be executed when an exception condition occurs. If a primary vector is declared, that condition handler is the first handler to be executed.

private section An image section of a process that is not sharable among processes. See also global section.

privilege See process privilege, user privilege, and image privilege.

privileged instructions In general, any instructions intended for use by the operating system or privileged system programs. In particular, instructions that the processor will not execute unless the current access mode is kernel mode (e.g., HALT, SVPCTX, LDPCTX, MTPR, and MFPR).

procedure 1. A routine entered via a call instruction. 2. See command procedure.

process The basic entity scheduled by the system software that provides the context in which an image executes. A process consists of an address space and both hardware and software context.

process address space See process space.

process context The hardware and software contexts of a process.

process control block (PCB) A data structure used to contain process context. The hardware PCB contains the hardware context. The software PCB contains the software context, which includes a pointer to the hardware PCB.

process header A data structure that contains the hardware PCB, accounting and quota information, process section table, working set list, and the page tables defining the virtual layout of the process.

process header slots That portion of the system address space in which the system stores the process headers for the processes in the balance set. The number of process header slots in the system determines the number of processes that can be in the balance set at any one time.

process identification (PID) The operating system's unique 32-bit binary value assigned to a process.

process I/O segment That portion of a process control region that contains the process permanent RMS internal file access block for each open file, and the I/O buffers, including the command interpreter's command buffer and command descriptors.

process name A 1- to 15-character ASCII string that can be used to identify processes executing under the same group number.

processor register A part of the processor used by the operating system software to control the execution states of the computer system. They include the system base and length registers, the program and control region base and length registers, the system control block base register, the software interrupt request register, and many more.

Processor Status Longword (PSL) A system programmed processor register consisting of a word of privileged processor status and the PSW. The privileged processor status information includes: the current IPL (interrupt priority level), the previous access mode, the current access mode, the interrupt stack bit, the trace trap pending bit, and the compatibility mode bit.

Processor Status Word (PSW) The low-order word of the Processor Status Longword. Processor status information includes: the condition codes (carry, overflow, zero, negative), the arithmetic trap enable bits (integer overflow, decimal overflow, floating underflow), and the trace enable bit.

process page tables The page tables used to describe process virtual memory.

process priority The priority assigned to a process for scheduling purposes. The operating system recognizes 32 levels of process priority, where 0 is low and 31 high. Levels 16 through 31 are used for time-critical processes. The system does not modify the priority of a time-critical process (although the system manager or process itself may). Levels 0 through 15 are used for normal processes. The system may temporarily increase the priority of a normal process based on the activity of the process.

process privileges The privileges granted to a process by the system, which are a combination of user privileges and image privileges. They include, for example, the privilege to: affect other processes associated with the same group as the user's group, affect any process in the system regardless of UIC, set process swap mode, create permanent event flag clusters, create another process, create a mailbox, perform direct I/O to a file-structured device.

process section See private section.

process space The lowest-addressed half of virtual address space, where per-process instructions and data reside. Process space is divided into a program region and a control region.

Program Counter (PC) General register 15 (R15). At the beginning of an instruction's execution, the PC normally contains the address of a location in memory from which the processor will fetch the next instruction it will execute.

program locality A characteristic of a program that indicates how close or far apart the references to locations in virtual memory are over time. A program with a high degree of locality does not refer to many widely scattered virtual addresses in a short period of time.

programmer number See member number.

program region The lowest-addressed half of process address space (P0 space). The program region contains the image currently being executed by the process and other user code called by the image.

Program region Base Register (P0BR) The processor register, or its equivalent in a hardware process control block, that contains the base virtual address of the page table entry for virtual page number 0 in a process program region.

Program region Length Register (P0LR) The processor register, or its equivalent in a hardware process control block, that contains the number of entries in the page table for a process program region.

program section (psect) A portion of a program with a given protection and set of storage management attributes. Program sections that have the same attributes are gathered together by the linker to form an image section.

project number See group number or account number.

pure code See reentrant code.

quadword Eight contiguous bytes (64 bits) starting on an addressable byte boundary. Bits are numbered from right to left, 0 to 63. A quadword is identified by the address of the byte containing the low-order bit (bit 0). When interpreted arithmetically, a quadword is a 2's complement integer with significance increasing from bit 0 to bit 62. Bit 63 is used as the sign bit. The value of the integer is in the range -2^{63} to $2^{63}-1$.

qualifier A portion of a command string that modifies a command verb or command parameter by selecting one of several options. A qualifier, if present, follows the command verb or parameter to which it applies and is in the format: "/qualifier:option". For example, in the command string "PRINT filename /COPIES:3", the COPIES qualifier indicates that the user wants three copies of a given file printed.

queue 1. n. A circular, doubly-linked list. See system queues. v. To make an entry in a list or table, perhaps using the INSQUE instruction. 2. See job queue.

queue priority The priority assigned to a job placed in a spooler queue or a batch queue.

quota The total amount of a system resource, such as CPU time, that a job is allowed to use in an accounting period, as specified by the system manager in the user authorization file. See also limit.

random access by record's file address The retrieval of a record by its unique address, which is provided to the program by RMS. This method of access can be used to randomly access a sequentially organized file containing variable length records.

random access by relative record number The retrieval or storage of a record by specifying its position relative to the beginning of the file.

read access type An instruction or procedure operand attribute indicating that the specified operand is only read during instruction or procedure execution.

record access block (RAB) An RMS user data structure that represents a request for a record access stream. A RAB relates to operations on the records within a file, such as UPDATE, DELETE, or GET.

record access mode The method used in RMS for retrieving and storing record in a file. One of three methods: sequential, random, and record's file address.

record cell A fixed-length area in a relatively organized file that is used to contain one record.

Record Management Services A set of operating system system procedures that are called by programs to process files and records within files. RMS allows programs to issue READ and WRITE requests at the record level (record I/O) as well as read and write blocks (block I/O). RMS is an integral part of the system software. RMS procedures run in executive mode.

record-oriented device A device such as a terminal, line printer, or card reader, on which the largest unit of data a program can access in one I/O operation is the device's physical record.

record's file address The unique address of a record in a file that allows records to be accessed randomly regardless of file organization.

record slot A fixed length area in a relatively organized file that is used to contain one record.

reentrant code Code that is never modified during execution. It is possible to let many users share the same copy of a procedure or program written as reentrant code.

register A storage location in hardware logic other than main memory. See also general register, processor register, and device register.

register deferred indexed mode An indexed addressing mode in which the base operand specifier uses register deferred mode addressing.

register deferred mode In register deferred mode addressing, the contents of the specified register are used as the address of the actual instruction operand.

register mode In register mode addressing, the contents of the specified register are used as the actual instruction operand.

relative file organization A file organization in which the file contains fixed length record slots. Each slot is assigned a consecutive number that represents its position relative to the beginning of a file. Records within each slot can be as big as or smaller than the slot. Relative file organization permits sequential record access, random record access by record number, and random record access by record's file address.

resource A physical part of the computer system such as a device or memory, or an interlocked data structure such as a mutex. Quotas and limits control the use of physical resources.

resource wait mode An execution state in which a process indicates that it will wait until a system resource becomes available when it issues a service request requiring a resource. If a process wants notification when a resource is not available, it can disable resource wait mode during program execution.

return status code See status code.

Run Time Procedure Library The collection of procedures available to native mode images at run time. These library procedures (such as trigonometric functions, etc.) are common to all native mode images, regardless of the language processor used to compile or assemble the program.

scatter/gather The ability to transfer in one I/O operation data from discontiguous pages in memory to contiguous blocks on disk, or data from contiguous blocks on disk to discontiguous pages in memory.

secondary storage Random access mass storage.

secondary vector A location that identifies the starting address of a condition handler to be executed when a condition occurs and the primary vector contains zero or the handler to which the primary vector points chooses not to handle the condition.

section A portion of process virtual memory that has common memory management attributes (protection, access, cluster factor, etc.). It is created from an image section, a disk file, or as the result of a Create Virtual Address Space system service. See global section, private section, image section, and program section.

sequential access mode The retrieval or storage of records in which a program successively reads or writes records one after the other in the order in which they appear, starting and ending at any arbitrary point in the file.

sequential file organization A file organization in which records appear in the order in which they were originally written. The records can be fixed length or variable length. Although one does not speak of record slots with sequentially organized files, for purposes of comparison with relatively organized files one can say that the record itself is the same as its record slot, and its record number is the same as its relative slot number. Sequential file organization permits sequential record access and random access by record's file address. Sequential file organization with fixed length records also permits random access by relative record number.

sharable image An image that has all of its internal references resolved, but which must be linked with an object module(s) to produce an executable image. A sharable image cannot be executed. A sharable image file can be used to contain a library of routines. A sharable image can be used to create a global section by the system manager.

shell process A predefined process that the job initiator copies to create the minimum context necessary to establish a process.

signal 1. An electrical impulse conveying information. 2. The software mechanism used to indicate that an exception condition was detected.

slave terminal A terminal from which it is not possible to issue commands to the command interpreter. A terminal assigned to application software.

small process A system process that has no control region in its virtual address space and has an abbreviated context. Examples are the working set swapper and the null process. A small process is scheduled in the same manner as user processes, but must remain resident during its execution.

software context The context maintained by the operating system that describes a process. See software process control block (PCB).

software interrupt An interrupt generated on interrupt priority level 1 through 15, which can be requested only by software.

software process control block (PCB) The data structure used to contain a process' software context. The operating system defines a software PCB for every process when the process is created. The software PCB includes the following kinds of information about the process: current state; storage address if it is swapped out of memory; unique identification of the process, and address of the process header (which contains the hardware PCB). The software PCB resides in system region virtual address space. It is not swapped with a process.

software priority See process priority and queue priority.

spooling Output spooling: The method by which output to a low-speed peripheral device (such as a line printer) is placed into queues maintained on a high-speed device (such as disk) to await transmission to the low-speed device. Input spooling: The method by which input from a low-speed peripheral (such as the card reader) is placed into queues maintained on a high-speed device (such as disk) to await transmission to a job processing that input.

spool queue The list of files supplied by processes that are to be processed by a symbiont. For example, a line printer queue is a list of files to be printed on the line printer.

stack An area of memory set aside for temporary storage, or for procedure and interrupt service linkages. A stack uses the last-in, first-out concept. As items are added to ("pushed on") the stack, the stack pointer decrements. As items are retrieved from ("popped off") the stack, the stack pointer increments.

stack frame A standard data structure built on the stack during a procedure call, starting from the location addressed by the FP to lower addresses, and popped off during a return from procedure. Also called call frame.

Stack Pointer General register 14 (R14). SP contains the address of the top (lowest address) of the processor-defined stack. Reference to SP will access one of the five possible stack pointers, kernel, executive, supervisor, user, or interrupt, depending on the value in the current mode and interrupt stack bits in the Processor Status Longword (PSL).

state queue A list of processes in a particular processing state. The scheduler uses state queues to keep track of processes' eligibility to execute. They include: processes waiting for a common event flag, suspended processes, and executable processes.

status code A longword value that indicates the success or failure of a specific function. For example, system services always return a status code in R0 upon completion.

store through See write through.

strong definition Definition of a global symbol that is explicitly available for reference by modules linked with the module in which the definition occurs. The linker always lists a global symbol with a strong definition in the symbol portion of the map. The librarian always includes a global symbol with a strong definition in the global symbol table of a library.

strong reference A reference to a global symbol in an object module that requests the linker to report an error if it does not find a definition for the symbol during linking. If a library contains the definition, the linker incorporates the library module defining the global symbol into the image containing the strong reference.

subprocess A subsidiary process created by another process. The process that creates a subprocess is its owner. A subprocess receives resource quotas and limits from its owner. When an owner process is removed from the system, all its subprocesses (and their subprocesses) are also removed.

supervisor mode The third most privileged processor access mode (mode 2). The operating system's command interpreter runs in supervisor mode.

suspension A state in which a process is inactive, but known to the system. A suspended process becomes active again only when another process requests the operating system to resume it. Contrast with hibernation.

swap mode A process execution state that determines the eligibility of a process to be swapped out of the balance set. If process swap mode is disabled, the process working set is locked in the balance set.

swapping The method for sharing memory resources among several processes by writing an entire working set to secondary storage (swap out) and reading another working set into memory (swap in). For example, a process' working set can be written to secondary storage while the process is waiting for I/O completion on a slow device. It is brought back into the balance set when I/O completes. Contrast with paging.

switch See (command) qualifier.

symbiont A full process that transfers record-oriented data to or from a mass storage device. For example, an input symbiont transfers data from card readers to disks. An output symbiont transfers data from disks to line printers.

symbiont manager The function (in the system process called the job controller) that maintains spool queues, and dynamically creates symbiont processes to perform the necessary I/O operations.

symbol See local symbol, global symbol, and universal global symbol.

Synchronous Backplane Interconnect (SBI) The part of the hardware that interconnects the processor, memory controllers, MASSBUS adaptors, the UNIBUS adaptor.

synchronous record operation A mode of record processing in which a user program issues a record read or write request and then waits until that request is fulfilled before continuing to execute.

system In the context "system, owner, group, world," the system refers to the group numbers that are used by operating system and its controlling users, the system operators and system manager.

system address space See system space and system region.

System Base Register (SBR) A processor register containing the physical address of the base of the system page table.

system buffered I/O An I/O operation, such as terminal or mailbox I/O, in which an intermediate buffer from the system buffer pool is used instead of a process-specified buffer. Contrast with direct I/O.

System Control Block (SCB) The data structure in system space that contains all the interrupt and exception vectors known to the system.

System Control Block Base register (SCBB) A processor register containing the base address of the system control block.

system device The random access mass storage device unit on which the volume containing the operating system software resides.

system dynamic memory Memory reserved for the operating system to allocate as needed for temporary storage. For example, when an image issues an I/O request, system dynamic memory is used to contain the I/O request packet. Each process has a limit on the amount of system dynamic memory that can be allocated for its use at one time.

System Identification Register A processor register which contains the processor type and serial number.

system image The image that is read into memory from secondary storage when the system is started up.

System Length Register (SLR) A processor register containing the length of the system page table in longwords, that is, the number of page table entries in the system region page table.

System Page Table (SPT) The data structure that maps the system region virtual addresses, including the addresses used to refer to the process page tables. The system page table (SPT) contains one page table entry (PTE) for each page of system region virtual memory. The physical base address of the SPT is contained in a register called the SBR.

system process A process that provides system-level functions. Any process that is part of the operating system. See also small process, fork process.

system programmer A person who designs and/or writes operating systems, or who designs and writes procedures or programs that provide general purpose services for an application system.

system queue A queue used and maintained by operating system procedures. See also state queues.

system region The third quarter of virtual address space. The lowest-addressed half of system space. Virtual addresses in the system region are sharable between processes. Some of the data structures mapped by system region virtual addresses are: system entry vectors, the system control block (SCB), the system page table (SPT), and process page tables.

system services Procedures provided by the operating system that can be called by user processes.

system space The highest-addressed half of virtual address space. See also system region.

system virtual address A virtual address identifying a location mapped by an address in system space.

system virtual space See system space.

task An RSX-11/IAS term for a process and image bound together.

terminal The general name for those peripheral devices that have keyboards and video screens or printers. Under program control, a terminal enables people to type commands and data on the keyboard and receive messages on the video screen or printer. Examples of terminals are the LA36 DECwriter hard-copy terminal and VT52 video display terminal.

time-critical process A process assigned to a software priority level between 16 and 31, inclusive. The scheduling priority assigned to a time-critical process is never modified by the scheduler, although it can be modified by the system manager or process itself.

timer A system fork process that maintains the time of day and the date. It also scans for device timeouts and performs time-dependent scheduling upon request.

track A collection of blocks at a single radius on one recording surface of a disk.

transfer address The address of the location containing a program entry point (the first instruction to execute).

translation buffer An internal processor cache containing translations for recently used virtual addresses.

trap An exception condition that occurs at the end of the instruction that caused the exception. The PC saved on the stack is the address of the next instruction that would normally have been executed. All software can enable and disable some of the trap conditions with a single instruction.

trap enables Three bits in the Processor Status Word that control the processor's action on certain arithmetic exceptions.

two's complement A binary representation for integers in which a negative number is one greater than the bit complement of the positive number.

two-way associative cache A cache organization which has two groups of directly mapped blocks. Each group contains several blocks for each index position in the cache. A block of data from main memory can go into any group at its proper index position. A two-way associative cache is a compromise between the extremes of fully associative and direct mapping cache organizations that takes advantage of the features of both.

type ahead A terminal handling technique in which the user can enter commands and data while the software is processing a previously entered command. The commands typed ahead are not echoed on the terminal until the command processor is ready to process them. They are held in a type ahead buffer.

unit record device A device such as a card reader or line printer.

universal global symbol A global symbol in a sharable image that can be used by modules linked with that sharable image. Universal global symbols are typically a subset of all the global symbols in a sharable image. When creating a sharable image, the linker ensures that universal global symbols remain available for reference after symbols have been resolved.

unwind the call stack To remove call frames from the stack by tracing back through nested procedure calls using the current contents of the FP register and the FP register contents stored on the stack for each call frame.

urgent interrupt An interrupt received on interrupt priority levels 24 through 31. These can be generated only by the processor for the interval clock, serious errors, and power fail.

user authorization file A file containing an entry for every user that the system manager authorizes to gain access to the system. Each entry identifies the user name, password, default account, User Identification Code (UIC), quotas, limits, and privileges assigned to individuals who use the system.

user environment test package (UETP) A collection of routines that verify that the hardware and software systems are complete, properly installed, and ready to be used.

User File Directory (UFD) See directory.

User Identification Code (UIC) The pair of numbers assigned to users and to files, global sections, common event flag clusters, and mailboxes that specifies the type of access (read and/or write access, and in the case of files, execute and/or delete access) available to the owners, group, world, and system. It consists of a group number and a member number separated by a comma.

user mode The least privileged processor access mode (mode 3). User processes and the Run Time Library procedures run in user mode.

user name The name that a person types on a terminal to log on to the system.

user number See member number.

user privileges The privileges granted a user by the system manager. See process privileges.

utility A program that provides a set of related general purpose functions, such as a program development utility (an editor, a linker, etc.), a file management utility (file copy or file format translation program), or operations management utility (disk backup/restore, diagnostic program, etc.).

value return registers The general registers R0 and R1 used by convention to return function values. These registers are not preserved by any called procedures. They are available as temporary registers to any called procedure. All other registers (R2, R3,..., R11, AP, FP, SP, PC) are preserved across procedure calls.

variable-length bit field A set of zero to 32 contiguous bits located arbitrarily with respect to byte boundaries. A variable bit field is specified by four attributes: 1) the address A of a byte, 2) the bit position P of the starting location of the bit field with respect to bit 0 of the byte at address A, 3) the size, in bits, of the bit field, and 4) whether the field is signed or unsigned.

variable-length record format A file format in which records are not necessarily the same length.

variable with fixed-length control (VFC) record format A file format in which records of variable length contain an additional fixed-length control area. The control area may be used to contain file line numbers and/or print format controls.

vector 1. A interrupt or exception vector is a storage location known to the system that contains the starting address of a procedure to be executed when a given interrupt or exception occurs. The system defines separate vectors for each interrupting device controller and for classes of exceptions. Each system vector is a longword. 2. For the purposes of exception handling, users can declare up to two software exception vectors (primary and secondary) for each of the four access modes. Each vector contains the address of a condition handler. 3. A one-dimensional array.

version number 1. The field following the file type in a file specification. It begins with a period (.) and is followed by a number which generally identifies it as the latest file created of all files having the identical file specification but for version number. 2. The number used to identify the revision level of program.

virtual address A 32-bit integer identifying a byte "location" in virtual address space. The memory management hardware translates a virtual address to a physical address. The term virtual address may also refer to the address used to identify a virtual block on a mass storage device.

virtual address space The set of all possible virtual addresses that an image executing in the context of a process can use to identify the location of an instruction or data. The virtual address space seen by the programmer is a linear array of 4,294,967,296 (2^{32}) byte addresses.

virtual block A block on a mass storage device referred to by its file-relative address rather than its logical (volume-oriented) or physical (device-oriented) address. The first block in a file is always virtual block 1.

virtual I/O functions A set of I/O functions that must be interpreted by an ancillary control process.

virtual memory The set of storage locations in physical memory and on disk that are referred to by virtual addresses. From the programmer's viewpoint, the secondary storage locations appear to be locations in physical memory. The size of virtual memory in any system depends on the amount of physical memory available and the amount of disk storage used for non-resident virtual memory.

virtual page number The virtual address of a page of virtual memory.

volume A mass storage medium such as a disk pack or reel of magnetic tape.

volume set The file-structured collection of data residing on one or more mass storage media.

wait To become inactive. A process enters a process wait state when the process suspends itself, hibernates, or declares that it needs to wait for an event, resource, mutex, etc.

wake To activate a hibernating process. A hibernating process can be awakened by another process or by the timer process, if the hibernating process or another process scheduled a wake-up call.

weak definition Definition of a global symbol that is not explicitly available for reference by modules linked with the module in which the definition occurs. The librarian does not include a global symbol with a weak definition in the global symbol table of a library. Weak definitions are often used when creating libraries to identify those global symbols that are needed only if the module containing them is otherwise linked with a program.

weak reference A reference to a global symbol that requests the linker not to report an error or to search the default library's global symbol table to resolve the reference if the definition is not in the modules explicitly supplied to the linker. Weak references are often used when creating object modules to identify those global symbols that may not be needed at run time.

wild card A symbol, such as an asterisk, that is used in place of a file name, file type, directory name, or version number in a file specification to indicate "all" for the given field.

window See mapping window.

word Two contiguous bytes (16 bits) starting on an addressable byte boundary. Bits are numbered from the right, 0 through 15. A word is identified by the address of the byte containing bit 0. When interpreted arithmetically, a word is a 2's complement integer with significance increasing from bit 0 to bit 14. If interpreted as a signed integer, bit 15 is the sign bit. The value of the integer is in the range -32768 to 32767. When interpreted as an unsigned integer, significance increases from bit 0 through bit 15 and the value of the unsigned integer is in the range 0 through 65535.

working set The set of pages in process space to which an executing process can refer without incurring a page fault. The working set must be resident in memory for the process to execute. The remaining pages of that process, if any, are either in memory and not in the process working set or they are on secondary storage.

working set swapper A system process that brings process working sets into the balance set and removes them from the balance set.

world In the context "system, owner, group, world," world refers to all users, including the system operators, the system manager, and users both in an owner's group and in any other group.

write access type The specified operand of an instruction or procedure is written only during that instruction's or procedure's execution.

write allocate A cache management technique in which cache is allocated on a write miss as well as on the usual read miss.

write back A cache management technique in which data from a write operation to cache is copied into main memory only when the data in cache must be overwritten. This results in temporary inconsistencies between cache and main memory. Contrast with write through.

write through A cache management technique in which data from a write operation is copied in both cache and main memory. Cache and main memory data are always consistent. Contrast with write back.

COMMONLY USED MNEMONICS

ACP	Ancillary Control Process
ANS	American National Standard
ASCII	American Standard Code for Information Interchange
AST	Asynchronous System Trap
ASTLVL	Asynchronous System Trap level
CCB	Channel Control Block
CM	Compatibility Mode bit in the hardware PSL
CRB	Channel Request Block
CRC	Cyclic Redundancy Check
DAP	Data Access Protocol
DDB	Device Data Block
DDCMP	DIGITAL Data Communications Message Protocol
DDT	Driver Data Table
DV	Decimal Overflow trap enable bit in the PSW
ECB	Exit Control Block
ECC	Error Correction Code
ESP	Executive Mode Stack Pointer
ESR	Exception Service Routine
F11ACP	Files-11 Ancillary Control Process
FAB	File Access Block
FCA	Fixed Control Area
FCB	File Control Block
FCS	File Control Services
FDT	Function Decision Table
FP	Frame pointer
FPD	First Part (of an instruction) Done
FU	Floating Underflow trap enable bit in the PSW
GSD	Global Section Descriptor
GST	Global Symbol Table
IDB	Interrupt Dispatch Block
IPL	Interrupt Priority Level
IRP	I/O Request Packet
ISECT	Image Section
ISD	Image Section Descriptor
ISP	Interrupt Stack Pointer
IS	Interrupt Stack bit in PSL
ISR	Interrupt Service Routine
IV	Integer Overflow trap enable bit in the PSW
KSP	Kernel Mode Stack Pointer
MBA	MASSBUS Adapter
MBZ	Must Be Zero
MCR	Monitor Console Routine
MFD	Master File Directory
MFPR	Move From Process Register instruction
MME	Memory Mapping Enable
MTPR	Move To Process Register instruction
MUTEX	Mutual Exclusion semaphore
NSP	Network Services Protocol
OPCOM	Operator Communication Manager
P0BR	Program region base register
P0LR	Program region length register
P0PT	Program region page table
P1BR	Control region base register
P1LR	Control region limit register
P1PT	Control region page table
PC	Program Counter
PCB	Process Control Block
PCBB	Process Control Block Base register
PFN	Page Frame Number
PID	Process Identification Number
PME	Performance Monitor Enable bit in PCB
PSECT	Program Section
PSL	Processor Status Longword
PSW	Processor Status Word
PTE	Page Table Entry
QIO	Queue Input/Output Request system service
RAB	Record Access Block
RFA	Record's File Address
RMS	Record Management Services
RWED	Read, Write, Execute, Delete
SBI	Synchronous Backplane Interconnect
SBR	System Base Register
SCB	System Control Block
SCBB	System Control Block Base register
SLR	System Length Register
SP	Stack Pointer
SPT	System Page Table
SSP	Supervisor Mode Stack Pointer
SVA	System virtual address
TP	Trace trap Pending bit in PSL
UBA	UNIBUS Adapter
UCB	Unit Control Block
UETP	User Environment Test Package
UFD	User File Directory
UIC	User Identification Code
USP	User Mode Stack Pointer
VCB	Volume Control Block
VPN	Virtual Page Number
WCB	Window Control Block
WCS	Writable Control Store
WDCS	Writable Diagnostic Control Store

BLANK

Index

BLANK

digital

DIGITAL EQUIPMENT CORPORATION, Corporate Headquarters: Maynard, Massachusetts 01754, Telephone: (617)897-5111—SALES AND SERVICE OFFICES: UNITED STATES—ALABAMA, Huntsville • ARIZONA, Phoenix and Tucson • CALIFORNIA, El Segundo, Los Angeles, Oakland, Ridgecrest, San Diego, San Francisco (Mountain View), Santa Ana, Santa Clara, Stanford, Sunnyvale and Woodland Hills • COLORADO, Englewood • CONNECTICUT, Fairfield and Meriden • DISTRICT OF COLUMBIA, Washington (Lanham, MD) • FLORIDA, Ft. Lauderdale and Orlando • GEORGIA, Atlanta • HAWAII, Honolulu • ILLINOIS, Chicago (Rolling Meadows) • INDIANA, Indianapolis • IOWA, Bettendorf • KENTUCKY, Louisville • LOUISIANA, New Orleans (Metairie) • MARYLAND, Odenton • MASSACHUSETTS, Marlborough, Waltham and Westfield • MICHIGAN, Detroit (Farmington Hills) • MINNESOTA, Minneapolis • MISSOURI, Kansas City (Independence) and St. Louis • NEW HAMPSHIRE, Manchester • NEW JERSEY, Cherry Hill, Fairfield, Metuchen and Princeton • NEW MEXICO, Albuquerque • NEW YORK, Albany, Buffalo (Cheektowaga), Long Island (Huntington Station), Manhattan, Rochester and Syracuse • NORTH CAROLINA, Durham/Chapel Hill • OHIO, Cleveland (Euclid), Columbus and Dayton • OKLAHOMA, Tulsa • OREGON, Eugene and Portland • PENNSYLVANIA, Allentown, Philadelphia (Bluebell) and Pittsburgh • SOUTH CAROLINA, Columbia • TENNESSEE, Knoxville and Nashville • TEXAS, Austin, Dallas and Houston • UTAH, Salt Lake City • VIRGINIA, Richmond • WASHINGTON, Bellevue • WISCONSIN, Milwaukee (Brookfield) • INTERNATIONAL—ARGENTINA, Buenos Aires • AUSTRALIA, Adelaide, Brisbane, Canberra, Melbourne, Perth and Sydney • AUSTRIA, Vienna • BELGIUM, Brussels • BOLIVIA, La Paz • BRAZIL, Rio de Janeiro and Sao Paulo • CANADA, Calgary, Edmonton, Halifax, London, Montreal, Ottawa, Toronto, Vancouver and Winnipeg • CHILE, Santiago • DENMARK, Copenhagen • FINLAND, Helsinki • FRANCE, Lyon, Grenoble and Paris • GERMAN FEDERAL REPUBLIC, Cologne, Frankfurt, Hamburg, Hannover, Munich, Nuremburg, Stuttgart and West Berlin • HONG KONG • INDIA, Bombay • INDONESIA, Djakarta • IRELAND, Dublin • ITALY, Milan, Rome and Turin • IRAN, Tehran • JAPAN, Osaka and Tokyo • MALAYSIA, Kuala Lumpur • MEXICO, Mexico City • NETHERLANDS, Utrecht • NEW ZEALAND, Auckland and Christchurch • NORWAY, Oslo • PUERTO RICO, Santurce • SINGAPORE • SPAIN, Madrid • SWEDEN, Gothenburg and Stockholm • SWITZERLAND, Geneva and Zurich • UNITED KINGDOM, Birmingham, Bristol, Epsom, Edinburgh, Leeds, Leicester, London, Manchester and Reading • VENEZUELA, Caracas •

PRINTED IN U.S.A. EA 15963 20/78 11- 14 30